The BLISS Programming Language: A History

Ronald F. Brender
Compaq Computer Corporation*

August 8, 2000

Abstract

The BLISS programming language was invented by Bill Wulf and others at Carnegie-Mellon University in 1969, originally for the DEC PDP-10. BLISS-10 (as it came to be called) caught the interest of Ron Brender of Digital Equipment Corporation in 1970. After several years of collaboration, including the creation of BLISS-11 for the PDP-11, BLISS was adopted as Digital's official implementation language for use on its new line of VAX computers in 1975. Digital developed a completely new generation of BLISSes for the VAX, PDP-10 and PDP-11, which became widely used at Digital during the 1970s and 1980s. With the creation of the Alpha architecture in the early 1990s, BLISS expanded again to support it, in both 32- and 64-bit flavors. In 1995, BLISS began to support the Intel IA-32 architecture.

BLISS has a number of unusual characteristics, some of which have been both strengths and weaknesses over the years: it is typeless, requires use of an explicit *contents of* operator (written as a period or "dot"), takes an algorithmic approach to data structure definition, requires and provides little or no run-time support (not even I/O), has no `goto`, has an unusually rich compile-time language, and provides flexible linkages between calling and called routines.

This paper reviews the evolution and use of BLISS over its three decade lifetime. While the language is clearly near the end of its life, it is actually continuing to expand to new achitectures and organizations, though only as a captive tool.

*Formerly Digital Equipment Corporation during most of the period described.

© 2000 Compaq Computer Corporation

Contents

1 Introduction **1**

1.1 The BLISS Epochs 2

2 The CMU Years: 1969-1974 **3**

2.1 BLISS Goals in Context 6

2.2 What Makes BLISS Interesting 7

2.2.1 A Typeless Language 7

2.2.2 Storage Names and the Dot Operator 8

2.2.3 Algorithmic Data Structure Definition 11

2.2.4 Remapping of Storage 13

2.2.5 No `goto` 15

2.2.6 An Expression Language 16

2.2.7 No Run-Time Library 16

2.2.8 System Integration 20

2.3 BLISS-10 Implementation and Use 22

2.4 BLISS for the PDP-11 23

2.4.1 Hardware Differences 24

2.4.2 Language Improvements 26

2.5 BLISS-11 Implementation and Use 34

3 The Second Generation of BLISSes: 1975-1978 **37**

3.1 Language Evolution 39

3.1.1 Hardware Matters 39

3.1.2 Portability 40

3.1.3 General Improvements 51

3.2 Building the Compilers 70

3.2.1 BLISS-32C 70

3.2.2 BLISS-36C 71

3.2.3 BLISS-16C 73

3.2.4 Bootstrapping and Native Compilers 73

3.2.5 BLISS-32 73

3.2.6 BLISS-36 75

3.2.7 BLISS-16 76

3.3 Methodology 76

3.4 To Type Data or not to Type Data 77

3.5 The New Generation is Established 78

4 The Golden Years at DEC: 1978-1990 **79**

4.1 "BLISS Extensions" 79

4.2 Compile-Time Processing and VAX Ada 85

4.3 GEM 89

5 The MIPS Years: 1989-1993 91

6 The Alpha Years: 1990- 93

6.1 Language Evolution 93

6.1.1 BYTE, WORD, LONG and QUAD 94

6.1.2 Floating Point 94

6.1.3 Alignment and Granularity 96

6.1.4 BLISS-32 *and* BLISS-64 96

6.1.5 The BLISS-32/64 Hybrid 97

6.1.6 Tighter Checking 99

6.1.7 Alpha Linux Influences 100

6.2 Compiler Implementation 101

7 The Intel Years: 1995- 102

8 BLISS Futures 104

8.1 Non-Stop for Alpha 104

8.2 Windows NT for IA-64 104

9 Retrospective Assessment 107

9.1 Typelessness 107

9.2 Structures 108

9.3 Map and Bind Declarations 108

9.4 The Dot Operator 109

9.5 No Run-Time Library 110

9.6 No `goto` 110

9.7 Expression Language 111

9.8 Compiletime Language 111

9.9 Relationship to Optimization Technology 112

9.10 BLISS and Portability 112

9.11 Language Evolution 113

9.12 C 113

10 Summary **115**

Addendum – What's In a Name

References

1 Introduction

In 1968, William A. Wulf joined the Computer Science Department at Carnegie-Mellon University, after earning the Ph.D. degree at the University of Virginia. Wulf wanted to write an operating system, among other things, but first he wanted a high level language in which to write it. When the Department acquired a DEC PDP-10, Wulf and others set about designing and implementing an appropriate language for it, which came to be known as BLISS[1]. The character of the result was influenced in part by Wulf's prior experience with Burroughs Extended Algol [8].

In early 1970, Ronald F. Brender joined the newly formed Research Department within Software Engineering at Digital Equipment Corporation, after earning the Ph.D. degree at the University of Michigan. His initial focus was evaluation of possible implementation languages for Digital's computer lines, including certainly the PDP-10 and hopefully also the 18-bit PDP-9 and PDP-15 series and the newly introduced 16-bit PDP-11. Brender's background included the academically successful MAD[2] language (and the rather unsuccessful MAD/I language) at the University of Michigan and a desire to avoid the error proneness so characteristic of assembly language. Languages of interest at that time included PL/I, PL360, BCPL, Pascal, AED (a proprietary language promoted by SofTech, Inc); through the grapevine, he also heard about Wulf and the BLISS work at CMU.

Later in 1970 Brender traveled to CMU to meet Wulf and learn more about BLISS. Gradually their mutual interests turned into a multiple year collaboration and initiated a line of BLISS language and compiler development that has now spanned three decades.

[1] Over the years, Wulf has steadfastly held that BLISS is not and never was an acronym. Undaunted, colleagues has offered suggestions to fill that gap. Perhaps most commonly reported is "Basic Language for the Implementation of System Software". Others have suggested "Beautiful Language..." or, perhaps inspired by the example of JOVIAL, "Bill's Language...".

[2] Michigan Algorithmic Decoder, sort of a hybrid of Algol and Fortran.

1.1 The BLISS Epochs

The history of BLISS can be divided into epochs which correspond to the organizational context in which BLISS language and compiler development took place, the technical challenges that motivated BLISS evolution, or both. The following epochs provide the framework for much of the presentation that follows:

- The CMU Years: 1969-1974
- The Second Generation of BLISSes: 1975-1978
- The Golden Years at DEC: 1978-1990
- The Alpha Years: 1990-
- The IA-32 Years: 1995-

Even as this paper is being written, BLISS is continuing to expand into new domains – these are discussed briefly next. The final major section, A Retrospective Assessment, seeks to evaluate those characteristics of BLISS that served it well, and not so well.

2 The CMU Years: 1969-1974

In corresponding with Wulf about the early influences that led up to BLISS, he responded as follows[43]:

> *In 1964 I was an instructor and de facto assistant director on the computer center at the University of Virginia. That year an extraordinary computer – the Burroughs B5000 – was delivered to the university.*
>
> *The B5000 (later updated to a B5500) was optimized to support programs written in Algol '60. But it was much more than that! At a time when other systems ran single programs in "batch" mode, the B5500 multitasked several programs. At a time that paging was essentially unknown, the B5500 hardware supported tagged pointers and automatic bounds checking, and a "presence bit" that allowed individual rows of an array, or subroutines of a program to be "paged out" to disk. Best, from my perspective, at a time when operating systems were written in assembly language, the B5500 operating system was written in ESPOL, a dialect of Algol '60.*
>
> *It was a fantastic playground for a 20-something aspiring computer scientist. We could easily modify the operating system, recompile it, and run a test – all in the space of 10-15 minutes. It was a opportunity unknown on other systems of the day. We experimented with scheduling algorithms for both processor and I/O; we experimented with automatic archiving and retrieval techniques; we built a time sharing system to support Basic; we measured properties of programs that influenced their behavior in a multi-tasking environment. We even modified the hardware – for example, changing the memory modules to interleave on the low address bit in order to reduce average memory latency by increasing parallelism. Later, a second processor was added to the system, and we had the chance to play with all the new challenges of multiprocessors.*
>
> *And, somehow, I also managed to complete a Ph.D. So, in 1968 I left UVa for a faculty position in Computer Science at Carnegie-Mellon University in Pittsburgh.*
>
> *At the time, CMU had two computers – a Univac 1108, and the experimental time-sharing system from IBM, the System 360 model 67. Both had huge operating systems written in assembly language. Both lived behind the glass walls of the computer center. Worse - neither was accessible by an assistant professor who wanted to play with the kinds of things that had been so much fun on the B5500. Thus I resolved to write an operating system (although at the time I didn't know what machine it would be for).*

But first, I wanted a language to write the system in – I didn't want to use assembly language. Reflecting back on the B5500/ESPOL experience, there was really only one thing that pinched, namely data structures. Algol (and ESPOL) supported only arrays; it had no records, no pointers, and no heap. Attempting to correct that deficiency led to most of the distinctive features of the original Bliss[sic].

But, before talking about these features, there were two people at CMU at the time that had a profound effect on me and on Bliss.

About this time Edsger Dijkstra at the Technische Hochschule Eindhoven in the Netherlands had completed the THE operating system – the first operating system that had been carefully layered as a set of increasingly capable machines, and one of the first examples of a "structured program". It was for the THE that Dijkstra and his students invented the famous "P" and "V" synchronization primitives. It was also this milieu from which emerged his famous short paper "Go To Statement Considered Harmful". In that paper he argued that the GOTO (1) could be used to create tangled, hard to understand, "spaghetti" control constructs, and (2) that any program that could be written with a GOTO could be written without one – and hence that it should be banned from programming languages.

One of Dijkstra's students, Nico Habermann, who had worked on THE, came to CMU – first as a visitor, and then as a faculty member. He brought with him the passion for rigor, for clean simplicity, and for structure that characterized Dijkstra's work. Nico was involved primarily as critic/foil for the Bliss design, but he implemented the first syntax analyzer for Bliss and helped design the process by which we bootstrapped Bliss into itself, so his impact was significant – including the choice of the name.

Gordon Bell was also at CMU at this time. Gordon was an early employee of Digital Equipment Corp. (DEC), and had been the lead engineer on the design of several of their computers – including the PDP-10, which was a highly popular computer among the ARPA-sponsored researchers at the time. Gordon brokered a deal for the CMU Computer Science Department to acquire a PDP-10 (all other machines were owned by the Computer Center). So the decision to target the first Bliss implementation to the PDP-10 was an easy one.

Back to the design of Bliss. As I noted, the subject of data structures was the one deficient aspect of ESPOL. Considerable contortion and obscure source code resulted from trying to encode more "interesting" data structures in an array. Thus, I wanted a powerful technique for defining data structures and encapsulating the definition in a way that hid the details of the implementation. I further wanted to define an arbitrary data structure – ones that might involve hashing, or list searching, or, well, just about anything.

Most languages now define data structures as a composition of a few basic types – arrays, records, etc. However, in the end, when an element of a structure is accessed, the compiler generates a sequence of code, an algorithm. The compositional definition of the structure is simply a stylized way of describing these algorithms! Why not, I reasoned, cut out the middleman? Why not define the data structure as a subroutine-like algorithm. That way any imaginable data structure could be defined!

That decision and feature of Bliss was the falling domino that provoked a series of other decisions:

- *The assignment statement was defined to allow an arbitrary expression on the left hand side as well as on the right. The expression on the left defined the address of the memory location to be set, and the right hand side defined the value to be stored.*
- *The language was defined to be an "expression language". That is, every construct – including those thought of as "statements" in other languages, was defined as an expression. Thus arbitrary program fragments could appear on either side of an assignment.*
- *Most (in)famously, in order to have the semantics of expressions be the same on both the left and right hand side of an assignment, we defined the semantics of an identifier to be the address of the variable. Always! To extract the value of a memory location one had to apply an operator, for which we chose the relatively inconspicuous period, or dot.*

The other distinctive feature of Bliss was the decision to eliminate the GOTO. This decision was provoked by Dijkstra's article and a general feeling that operating systems were complex systems that needed to be carefully structured, but not by any feeling of necessity.

2.1 BLISS Goals in Context

In the late 1960s and early 1970s, the use of high-level languages for systems programming was highly controversial. Most systems had very limited amounts of main memory; indeed, even so-called large systems generally had hardware maximum address space limitations of 16 or 18-bits of addressable memory – and no virtual memory support either. "Size is the goal" for both code and data often summarized one side of the debate. Systems programmers wanted lots of control over every instruction and every cycle and every bit of memory used.

The advocates for using high-level languages cited the potential for improved productivity and reduced error rates through the use of structured programming and data abstraction[3]. But the languages that seemed expressive enough for the job at the time, languages such as PL/I or Algol 68, were not viewed as suitably efficient. Arguments raged over whether it was even conceivable, let alone practical, for compilers to produce code that was within some small incremental percentage[4] of what could be achieved by good programmers.

Lots of experimentation was in progress throughout the computer industry seeking languages with the right blend of high-level features, low-level control, ease of compiling to efficient code, and improved optimization technology. The few then contemporary languages that fit these goals according to the earliest paper on BLISS [47] were EPL [29], Burroughs Extended Algol [8], PL360 [39] and BCPL [31, 35].

In designing BLISS, Wulf [41] set forth these objectives and constraints for the language.

- It was targeted toward system programs, that is, programs where the application domain is the computer as such.
- It assumed knowledgeable users with respect to training and judgement.
- It must provide high efficiency, with little or no required run-time support or distributed overhead, none of which is a substitute for programmer choice of good algorithms.
- It was expected that programs would be maintained, modified and enhanced over long periods of time by a succession of developers (the "turn-over" problem as it was called).
- It must cater to and support good structured programming concepts and practices.

[3] These motivations surely sound familiar even today, although today the vehicles usually have the words "object" or "object oriented" in them!

[4] Pick your favorite threshold. *No one* ever picked zero...

2.2 What Makes BLISS Interesting

BLISS has a number of characteristics that set it apart from most then contemporary and subsequent programming languages. The following are of particular interest:

- Typeless
- The dot (".") operator
- Algorithmic data structures
- Remapping of storage
- No `goto`
- An expression language
- No run-time library (RTL) required
- Control of routine calling conventions

In the sections that follow, examples are presented using the "second generation" syntax that was developed at Digital during the mid-1970s. The original BLISS-10 and BLISS-11 syntax is illustrated or described only when discussing the evolution in syntactic style that occurred during that period (see especially Section 3.1.3).

2.2.1 A Typeless Language

BLISS is a *typeless* language; that is, all data is manipulated in terms of the underlying machine *word* sized units. The word size is usually the same as the (integer or general purpose) register size, which may be a multiple of the addressable unit size.

On the PDP-10, memory and registers consisted of 36-bit words.

A word can represent anything: an integer, an address, a floating point value, a character, a packed structure, whatever. The choice of interpretation is determined by the operation that is applied to that data. In this regard, BLISS is much like BCPL [31], a contemporary with BLISS in origin.

2.2.2 Storage Names and the Dot Operator

In BLISS, the value of the name of a variable is the address of that variable–sometimes called its *lvalue*–regardless of whether the name occurs on the left or right hand side of an assignment. To fetch the contents of a variable, an explicit unary *contents of* operator needs to be applied. For example:

```
X = .Y + 2;
```

fetches the contents of variable Y, adds 2, and stores the result at address X.[5]

This "context independent" evaluation of names makes it easy to interpret expressions that occur on either side of an assignment. For example, in

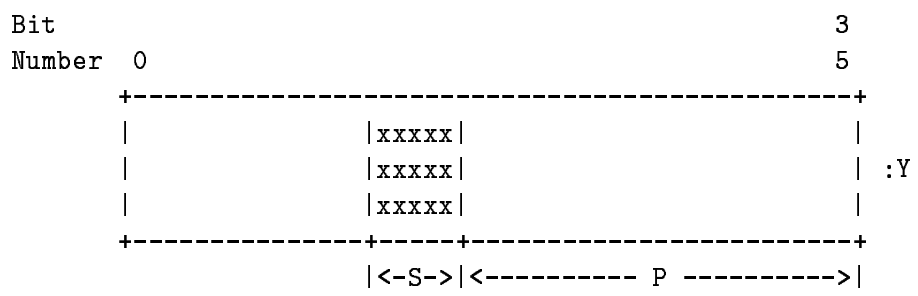
```
X+.I = .(X+.I) + 2
```

both occurrences of `X + .I` are interpreted identically: add the contents of `I` to the address of X, and use the resulting address as specified by the containing operators. Notice in particular the dot operator is applied again to the second occurrence of that expression, to fetch the contents at the computed address.

Of course, `X + .I` is easily recognized as an example of indexing, which leads naturally into the topic of structures. But first we complicate matters to understand partial word accesses.

The DEC PDP-10 [21] was a 36-bit word addressable machine with an 18-bit address space. It included Load Byte (LDB) and Deposit Byte (DPB) instructions for accessing that part of a word (not crossing word boundaries) characterized by a position P, the number of bits from the low-order end of the word, and a size S, the number of bits. The bits accessed are illustrated in Figure 1.

[5] DEC later adopted the term *data segment* to mean a region of memory, whether variable (read/write), constant, executable, local, global or whatever. Hereafter, data segment will generally be used unless the variableness of the segment is significant to the discussion.



where $0 \leq P \leq 36$, $0 \leq S \leq 36$, and $P + S \leq 36$.

Figure 1: PDP-10 Byte Access

To perform such accesses, the instructions operated indirectly through an operand called a *byte pointer* whose representation is illustrated in Figure 2. That is, it contained

- An 18-bit address (Y)
- A 4-bit index register name (X)
- A 1-bit indirection indicator (I)
- A 6-bit position value (P)
- A 6-bit size value (S)

BLISS-10 used the notation `Y<p, s, x, i>` to express a byte pointer. By default, the *undotted name* `Y` was equivalent to `Y<0, 36, 0`[6]`, 0>`, meaning the byte beginning at the lowest end of the word and extending for 36 bits, that is, the entire word.

[6] Register 0 was not used for indexing.

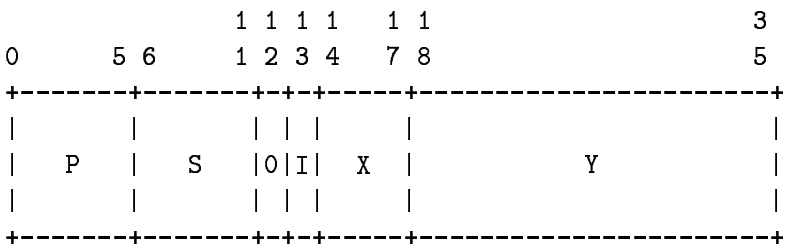


Figure 2: PDP-10 Byte Pointer

Actually, BLISS-10 had three contents operators:

- The dot as just described
- The back slash ("\") used the 18-bit address together with the index and indirection fields and assumed a full word position and size (0 and 36) for the contents operation.
- The at sign ("@") used only the address and assumed a full word position and size.

All three contents operators had the same effect if applied to a simple name; their differences only came into play when accessing the result of a more general expression, for example, another contents operation as in

```
X = ..Y;    ! Assumes Y contains a byte pointer
            ! (including non-zero P and S fields)
X = \.Y;    ! Assumes Y contains a byte pointer with
            ! non-zero indirection and indexing, but
            ! P and S ignored
X = @.Y;    ! Assumes Y contains an address, the rest
            ! of the contents is ignored
```

In practice, \ was almost never used. The @ operator did have some utility and use, but not a lot. (Some users suggested that the definitions of dot and at sign should have been reversed, on the grounds that the simpler fetch from address was the most common use. Others just used `@` all the time because they preferred a bigger and easier to see contents operator!)

```
structure
    VECTOR[I; N] =
        [N]
        VECTOR + I - 1;    ! 1-origin vector

global
    J,
    A : VECTOR[10];

J = 2;
A[.J] = .A[.J+1] + 3;
!
! The last assignment is equivalent to
!
! (A + (.J) - 1) = .(A + (.J+1) - 1) + 3;
```

Figure 3: BLISS Structure Example

2.2.3 Algorithmic Data Structure Definition

To understand the BLISS approach to data structures, consider Figure 3. The structure declaration defines VECTOR to be a 1-origin vector (as in FORTRAN) rather than a 0-origin vector (as in C). This structure has two kinds of formal parameters: the *access formals*, which precede the semicolon, and the *allocation formals*, which follow the semicolon.

After the equals sign (here meaning "is defined as") is an expression in square brackets called the *allocation expression*. This expression is used to compute the number of addressable storage units to allocate for a data item declared with that structure attribute. For example, the data segment A is declared with a VECTOR structure giving an *allocation actual* value of 10 – with the result that 10 words of storage are allocated for A and the name A corresponds to the first of them.

```
structure
    UPPERDIAG[I, J; M] =    ! M x M upper diagonal matrix
        [M*(M-1)/2]         ! Store only upper diagonal
        UPPERDIAG + (J-2)*(J-1)/2 + I - 1;

own
    B : UPPERDIAG[100];         ! Declare and allocate B

B[.K, 5] = .K[.K+1, .L];
!
! where the assignment is equivalent to
!
! T' = .K+1;
! ((B(.K-2)*(.K-1)/2+5 - 1) =
!     .((B+(.T'-2)*(.T'-2)/2 + .L -1);
```

Figure 4: BLISS Structure Upper Diagonal Structure Example

Following the allocation expression is the *access expression*, which is used when an element of the structure is accessed. The structure name itself acts as a kind of implicit formal that corresponds to the actual data segment. Thus the access `A[.J]` is semantically equivalent to `(A+.J-1)` – add the contents of J to the address of A and subtract one[7].

The expressive power of structures begins to emerge when considering an example such as shown in Figure 4, which declares and accesses an upper diagonal matrix. Notice in particular that the matrix is both declared and accessed just like an ordinary square matrix would be, without having to expose its internal organization.

It is perhaps less obvious how to declare and access non-algorithmic, record-like data segments. This can be accomplished by exploiting the byte pointer notation as part of a structure declaration, as illustrated in Figure 5. Notice that macro names are declared that define the word offset (`0`) and

[7]Note that the dot before J in the expansion is part of the access actual parameter, not the dot before the I in the structure definition. The semantics of the structure access expression is closer to that of a function body than a macro – thus the access expression is evaluated just once and `.I` refers to the contents of the implicit temporary where the actual value is saved. The implementation usually expands these "calls" inline, although closed form structure accesses can occur.

```
structure
    REC[O, P, S; N] =
        [N]
        (REC + O)<P, S>;

macro
    Word1LowHalf = 0,  0, 18 %,
    Word1Flag1   = 0, 18,  1 %,
    Word2Full    = 1,  0, 36 %;

own
    DATUM : REC[2];

DATUM[Word1LowHalf] = -1;
DATUM[Word1Flag1]   =  0;
DATUM[Word2Full]    =  1;
!
! where the assignments are equivalent to
!
!(DATUM + 0)<0, 18> = -1;
!(DATUM + 0)<18, 1> =  0;
!(DATUM + 1)<0, 36> =  1;
```

Figure 5: BLISS Record-like Structure Example

position and size within that word (P and S) for the component fields of a particular kind of record. These macros can then be used in structure accesses to provide a more mnemonic, record-like access notation.

2.2.4 Remapping of Storage

BLISS provides means to dynamically change the way that storage is viewed by using the map and bind declarations.

The simpler of the two is the map declaration which associates a new set of attributes with an already declared name. For example, suppose there were an inner scope where it was desirable to explicitly take advantage of the actual layout of the upper diagonal matrix B declared in the example of Figure 4. Such a scope might look like

```
begin

map
    B : VECTOR[100*99/2];       ! Same size as above

... B[.L] ...

end;
```

Within this block, the data segment B is accessed using the VECTOR structure, not the original UPPERDIAG structure.

More general is the bind declaration, which allows associating a name and attributes with a dynamically computed address. For example:

```
bind
    BV = MALLOC(100) : VECTOR[];
```

declares a new name BV and associates it with the value returned by calling the routine MALLOC. Note that the value of the undotted name is defined to be the function result, which in this case is thereafter intended to be used as the address of a vector. (The structure attribute in this case does not include allocation actual parameters because the attribute is not used to allocate storage.)

The map declaration can almost be expressed as a bind declaration as in

```
bind
    B = B : VECTOR[...];
```

The "almost" in this case is concerned solely with name visibility and scope matters that make this not quite legal, but nonetheless illustrative. A legal and more interesting variation might be

```
bind
    BV = B : VECTOR[...];
```

which allows the *same* storage to be viewed simultaneously as both an upper diagonal matrix and a vector[8].

Note that the bind declaration can be used in several ways:

- To provide a short hand name for a part of a larger aggregate.
- To provide a name to anonymous static storage that is allocated by other language mechanisms.
- To change the structure associated with an existing data segment (as discussed above).
- To capture the value of an arbitrary expression at a given point in time (entry to the block containing the declaration).

A more extreme example of remapping occurs when the same data segment is sometimes treated as data and sometimes executed as code, as illustrated in Figure 6. Here the occurrence of the data segment name on the left hand side of the assignment indicates that a storage assignment operation is to be performed, while later the () syntax indicates that a routine call operation is to be performed to that same data segment. This capability was later exploited to great advantage by DEC's APL-10 product which would create special case inline code for many APL operators —on the fly— and then call that code to evaluate an expression (rather than perform more traditional interpretation of expressions).

2.2.5 No goto

BLISS was the first commercial quality procedural language to heed Dijkstra's classic injunction about the harm of the goto [22] and not have one[9].

BLISS did include an unusually rich set of control structures, which are illustrated in Figures 7 and 8.

[8] Such aliasing presents all the obvious challenges to code optimization; however, discussion of this is beyond the scope of this paper.

[9] Only recently has another major language, Java [1], followed suit.

```
! Some data segment
!
own
    RTN : vector[1000];

! Treat as data to supply instructions
!
RTN[.I] = ...;

! Call as a routine to cause execution
!
RTN();
```

Figure 6: A Data Segment as Both Code and Data

A set of exit constructs allowed premature termination of the innermost containing blocks or control constructs (see further discussion in 2.4.2).

2.2.6 An Expression Language

BLISS is an *expression language*; that is, all of the control constructs are defined to yield usable values. The examples in Figure 9 are a bit contrived, but they illustrate the idea.

In the last example, `exitloop` (which exits the innermost containing loop) supplies the value of the `while` loop expression if it executes and the value is otherwise -1 if the loop runs to completion. (Minus one is an arbitrary but usually acceptable default value.)

2.2.7 No Run-Time Library

Since it was intended that BLISS be suitable for implementing all kinds of system software, including drivers and exception handlers that dealt with low-level hardware details, it was important that BLISS compiled code did not depend on the existence of a support library.

```
! Simple decisions
!
if .X gtr .Y then MAX = .X else MAX = .Y;

! Simple loops
!
do begin ...body of loop... end while .PTR neq 0;

while .PTR neq 0 do ...body of loop...;

X = 1; do X = .X*2 until .X gtr 100;

until .X gtr 100 do X = .X*2;

! Counted loops (control variable is local to loop body)
!
incr I from 1 to 10 do SUM = SUM + .ARR[.I];

decr J from .I to 0 by 2 do
    begin

    local
        ABS;

    if .ARR[.J] gtr 0 then ABS = .ARR[.J] else ABS = -.ARR[.J];
    ABSSUM = .ABSSUM + .ABS;

    end;
```

Figure 7: BLISS Control Flow Examples–Part 1

```
! Multiway decisions: uses a dispatch table
!
case F(.PTR) from 1 to 100 of
    set
    [1] :   ...case F(.PTR) eql 1...
    [3, 5, 7] :
            ...cases F(.PTR) eql 3, or 5, or 7...
    [inrange] :
            ...cases from 1 to 100 not otherwise handled
    [outrange] :
            ...cases outside range 1 to 100
    tes;

! Multiway decision: uses repeated comparisons
!
select .val of
    set

    [1] :   ...case 1...
    [1, 3] :...case 1 (again, both occurences match) or 3...
    ...and so on...

    tes;
```

Figure 8: BLISS Control Flow Examples–Part 2

```
! Assign last expression of block (.Y + .Z) to X
!
X = begin Y = ...; Z = ...; .Y + .Z end;

! Assign expression of executed alternative to M
!
MAX = if .A gtr .B then .A else .B;

! Assign value of exitloop (if executed) or -1 if
! loop terminates to CNT
!
P = .HEAD;
CNT = while .P neq 0 do
    begin
    if .(.P+4) eql 5 then exitloop .P+10;
    P = ..P;
    end;
```

Figure 9: Control Expression Values

Field	Function
opcode	A 9-bit code that specified the operation to be performed.
accumulator	A 4-bit accumulator field that specified one of the machine registers. The accumulator was either the source of one operand, where to place the result, or both, depending on the opcode.
address	An 18-bit address field.
index	A 4-bit index field that specified one of the registers whose contents is added to the address field to form the effective address.
indirection	A 1-bit field that specified whether to access indirectly through the address formed by the address and index fields.

Figure 10: PDP-10 Instruction Fields

2.2.8 System Integration

BLISS-10 provided features to integrate with a variety of hardware and operating system characteristics (beyond just those related to memory and addressing).

Machop Declarations

The PDP-10 instruction set encoding was so simple and regular that nearly any instruction could be invoked using a special function call; all that was required was to declare a suitable name and the value of the opcode to use in a machop declaration.

Every instruction consisted of five fields, as shown in Figure 10. Exploiting this uniformity, a declaration such as

```
machop
    OPENUUO = #050;          ! Octal 50
```

declared a special function named OPENUUO which could be called with up to four arguments, corresponding to the accumulator, address, index, and indirection. The *value* of such a call was uniformly taken to be the contents of the accumulator register after the operation is performed. (This made little sense in a few cases, but was adopted for uniformity.)

RESERVE=(r_1,...r_n)	Registers that are reserved. These registers were never allocated nor used by the generated code except as required by the source code. (A register declaration may specify the specific register in which the declared data segment is allocated. Such a register can then be fetched from and assigned to using normal BLISS notation.) Such registers were typically used for explicitly coded communication across routines and across modules.
SREG=s	The register to use as a stack pointer. (Register 17 (octal) was one popular choice.)
FREG=f	The register to use as a frame pointer (holding the value of the stack pointer on entry to a routine). The frame pointer was used to address parameters (which were pushed on the stack) and local variables of the current routine.
BREG=b	The register to use as the base of the stack for the current "thread". Needed to support the co-routine mechanism.
VREG=v	The register to use as the place to return the value of a routine call. (Register 0 was often used.)
DREGS=n	The number of registers to use as preserved registers across routine calls.

Figure 11: BLISS-10 Run-Time Register Controls

It was trivial to compile such a "call" into the appropriate PDP-10 instruction.

By this means, the entire PDP-10 instruction set became directly usable without needing to resort to an assembler tool.[10]

Run-Time Register Conventions

The PDP-10 software environment did not dictate a single integrated set of register usage conventions, so a number of alternatives had come into use over time. To be able to work with any of these, BLISS-10 included a number of compile time directives as shown in Figure 11.

The same values had to be used for all compilation units of a program, although there was no mechanism to enforce this.

[10] Such a general, low level mechanism could hardly go unabused. Several colleagues [24, 30] have reported knowing of or being themselves involved in attempts to use `machop` together with the PDP-10 `JUMPA` or `JRST` instructions to create a "fake" `goto` in BLISS–"just for fun", of course.

Other Controls

There were a variety of options for matching other environment characteristics (high versus low segment allocation, command processor interfacing, library entry points, and so on) which are beyond the scope of this paper.

2.3 BLISS-10 Implementation and Use

At the time Brender first met Wulf in 1970, the BLISS compiler for the PDP-10 was well along, though hardly complete. It had been successfully bootstrapped into itself[11]. While structures were used in its writing, these were not yet implemented; rather, structure references were implemented using a separate preprocessor written using TECO[12].

The compiler was (mostly) single pass, used recursive descent parsing and generated object code "on the fly". The code was not immediately emitted, however; rather it was buffered and subjected to various post-passes. The resulting code quality was reasonably good, but still allowed many opportunities for improvement. This compiler structure reflected the view that global optimization was part of the programmer's job of algorithm selection while best possible local code quality was the province of the compiler.

Brender joined the implementation effort as a means to better learn the language as well as to speed it completion. He added special case optimizations for various operators, designed and implemented a simple character oriented macro language, and generally took part in the continuing bootstrapping effort. There was no Internet in those days, even its predecessor ARPAnet was yet to emerge, so this collaboration involved frequent travel and many week long stays in Pittsburgh.

Use at CMU

BLISS easily attracted a following at CMU. It was used for SCRIBE (an early document formatting utility), PQCC (in the Production Quality Compiler Compiler research project), a WATFOR-like compiler, as well as a variety of other departmental and personal programming efforts.

[11] The compiler was initially written using BLISS as a sort of pseudo-code which was hand translated into assembly language, then the pseudo-code bootstrapped into itself.

[12] TECO was a very powerful character-oriented text editor of that period, almost a programming language in its own right in many ways. Structure definitions were scanned and parsed by one set of TECO macros which *compiled* them into another set of TECO macros. These latter TECO macros were then used to preprocess the BLISS sources, recognizing and expanding structure references into an equivalent form.

Use at DEC

As the BLISS compiler stabilized, Brender turned more attention toward introducing its use at DEC. As it happened, a new optimizing FORTRAN compiler development (known as FORTRAN-10) project was starting at a good time to use BLISS-10. As compiler and language oriented people themselves, the development team was eager to use a high level language for their work as well as being perhaps less risk averse than others regarding the strange new language from CMU. The FORTRAN-10 group plunged into using BLISS – and never looked back.

However, the PDP-10 was a popular and highly successful system used by timesharing service bureaus of that period; many of those customers very much wanted the new higher performance FORTRAN and did not want to risk it being implemented using BLISS. Before FORTRAN-10 was completed, the twice annual DECUS[13] meetings regularly sparkled with debate over the use of BLISS. At one especially memorable session, many of the critics showed up wearing bright lapel buttons that read "BLISS is ignorance"!

Fortunately, FORTRAN-10 did meet its goals and BLISS-10 was deemed to have contributed to that success.

BLISS became regularly used for new projects in the Large Computer Group and other parts of DEC. Besides FORTRAN-10, BLISS-10 was also used for BASIC-20[14], APL-10 and -20, MCS-10 (a message control system), RMS-20 (a record management facility similar in capability to that on the PDP-11 RSX-11M/D), as well as IDEA (a circuit board layout tool) and SAGE (a logic simulator).

2.4 BLISS for the PDP-11

Even as BLISS for the PDP-10 was becoming established, attention began to focus on creating a version of BLISS for the PDP-11, which became known as BLISS-11.

[13] Digital Equipment Corporation User's Society

[14] The -20 suffix indicates a product offered on the DECSYSTEM-20, a second operating system that ran on a variant of the PDP-10 hardware. The user mode software environment on the DECSYSTEM-20 was upward compatible with that on the DECsystem-10.

Property	PDP-11	PDP-10
Word size	16 bits	36 bits
Addressable unit	8-bit byte	36-bit word
Byte pointer	(just an) address	address<p, s, x, i>
Byte access	fetch sign extends	fetch zero extends
	(8-bit only)	(any size up to 36 bits)
Registers	8	16
	(special purpose: SP, PC)	(special purpose: AC 0)
Floating point	not available	used general registers

Figure 12: Summary of Differences: PDP-11 versus PDP-10

The differences between BLISS-11 and BLISS-10 are generally of two kinds:

- Those that result from differences in the underlying hardware
- Language improvements

These are discussed in turn in the following sections.

2.4.1 Hardware Differences

The important hardware differences (that influenced BLISS) between the PDP-11 and the PDP-10 are summarized in Figure 12.

Memory and Addressing

The word size (36 vs 16), addressable unit size (36 vs 8) and byte pointer (arbitrary vs 8-bit only) differences together meant that the BLISS-10 notion of a byte pointer could not be retained in BLISS-11: a 16-bit word could hold an address but no more. To accommodate this, the BLISS-10 <> operation was reformulated to become an optional part of the fetch and assignment syntax.

For example, in BLISS-11

```
Y<1, 1> = .X<0, 1>;
```

means the same as in BLISS-10: fetch the low order bit from location X and store it in the next to low bit of location Y. But

```
Y = X<3, 4>;
```

is not valid in BLISS-11.

To take advantage of byte addressability on the PDP-11 and allow convenient use of small data segments, BLISS-11 added the `BYTE` and `WORD` attributes which could be given on a data segment declaration; the default is `WORD`. These *allocation attributes* implied a default position and size of `<0, 8>` and `<0, 16>`, respectively, if no explicit specification was given. Thus, given

```
global
    P : BYTE,
    Q : WORD,
    R;
```

the following pairs are equivalent

```
Q = .P;     and         Q = .P<0, 8>;
P = .Q;     and         P<0, 8> = .Q;
R = .Q;     and     R<0, 16> = .Q<0, 16>;
```

A more subtle difference arose because the PDP-11 MOVB (move byte) instruction performed a sign extension when copying an 8-bit byte from memory into one of the 16-bit registers. It cost an extra instruction to clear the high order bits if zero extension was wanted. We were much concerned that defining the BLISS-11 fetch operation to always zero extend (to match BLISS-10) would generally be too expensive, especially for cases where sign extension might actually be preferred (because it was known a priori that possible values were limited in range to less than 128). But, neither could we justify sign extending subword sized fields of other than 8 bits. Ultimately, we retained zero-extension for all subword fetches and left it as a challenge to the BLISS-11 compiler to minimize the cost.

Address Comparisons

Address comparisons on the PDP-10 were always performed using signed relational instructions– the limited (18-bit) range of address values made signed and unsigned comparisons equivalent in practice. This was not the case on the PDP-11, where the word size and address size were the same (16-bits)[15].

For BLISS-11, we added unsigned versions of the relational operators, naming them after their BLISS-10 counterparts but with a suffix "U" appended (`EQLU`, `LEQU`, `LSSU`, and so on).

Floating Point

The PDP-10 supported single precision floating point with values held in the general registers; BLISS-10 provided binary infix operators FADR (add), FSBR (subtract), FMPR (multiply) and FDIV (divide), as well as the unary prefix operator FNEG (negate) to allow use of that capability.

The early PDP-11's provided no floating point hardware support, so none was included in BLISS-11 either. Even when floating point hardware options were later introduced, the floating point (32- and 64-bit) values did not fit in the 16-bit BLISS model and used a set of registers separate from the general purpose set. Because they did not fit the BLISS model, they were left out of BLISS-11.

Number of Registers

The difference in the number of registers was of little linguistic consequence.

2.4.2 Language Improvements

Significant language improvements were made in a number of areas, including most importantly:

- Macros
- Labels and exit forms
- Linkages
- File inclusion declaration (`require`)

[15] The importance of unsigned instead of signed comparisons for addresses immediately became embarrassingly apparent the first time DEC ran any of its system software on a PDP-11 with more than 32K bytes of memory!

Macros

The macro facility of BLISS-10 was very simple and, hence, also very limited. It allowed substitution of actual arguments for formal parameters during macro expansion, and little else. Because it was character stream oriented, various kinds of lexical oddities tended to interfere with routine use.

For BLISS-11, a completely new and much more functional macro mechanism was introduced. The three main forms are illustrated in Figures 13, 14 and 15.

Along with the macro mechanisms came a new notion: *lexical functions* that are completely evaluated during compilation. In the case of those illustrated in the Figures, these functions provide access to characteristics of the macro call and/or its expansion.

Other lexical functions such as `%name` and `%string`, which formed a name or a string, respectively, out of concatenation of their arguments, were also introduced. (This foreshadowed an explosion of compile time lexical mechanisms in later BLISSes).

Labels and Leave

It was mentioned earlier that BLISS-10 omitted the `goto` statement. It may seem sensible then that it should also omit labels. It did. And that was later seen as a mistake.

The utility of an early escape from a loop is widely appreciated – witness the loop exiting constructs of C, C++ and numerous other languages (for example, `break`). BLISS-10 carried this to the extreme of providing an escape form for every control flow construct in the language. The general form was:

escape-keyword [*level*] *expression*

The escape-keywords included:

- `exitblock`,
- `exitcompound`,
- `exitloop`,
- `exitcond` (exit one arm of a conditional, eg., `if-then-else`),
- `exit` (any control scope),

A simple macro expands once, substituting each actual parameter (possibly none) for the corresponding formal parameter. Recursion is not allowed. A macro without a formal list is called without an argument list.

Example Declarations:

```
macro
    LOWBYTE  = 0, 8 %,
    HIGHBYTE = 8, 8 %,
    OUT_ERROR(NUM, MSG) =
        OUTSTRING(plit(%string('ERR ', NUM, ':', MSG)) %;
```

Example Calls:

```
A<LOWBYTE> = .B<HIGHBYTE>;
OUT_ERROR(17, 'Ambiguous');
```

which expand to

```
A<0, 8> = .B<8, 8>;
OUTSTRING(plit('ERR 17: Ambiguous'));
```

Figure 13: BLISS-11 Simple Macros

A conditional macro expands if and only if there is at least one argument more than needed for the given formal parameters. Recursion is allowed.

Example Declaration:

```
macro
    EL[]  = else %,
    COND(BOOL, ACTION) =
        if (BOOL) then ACTION
            EL(%remaining) COND(%remaining) %;
```

where

- `%remaining` is a lexical function whose expansion is all remaining arguments included in the call.

Example Call:

```
COND(.A eql .B, X = FOO(), .A gtr .B, X = 0);
```

which expands to

```
if (.A eql .B) then X = FOO()
    else if (.A gtr .B) then X = 0;
```

Figure 14: BLISS-11 Conditional Macros

An iterated macro expands repeatedly, using the same actuals each time for the fixed formal parameters (declared within parentheses) and successive actual parameters corresponding to the iterated formals (declared in square brackets) until there are too few unused actuals to match the list of iterated formals. Recursion is allowed.

Example Declarations:

```
macro
    LOAD_VEC(NAME)[VALUE] =
        NAME[%count] = VALUE ! Note: no final ;
        %,
    INIT_LOCAL_VEC(NAME)[] =
        local
            NAME : VECTOR[%length-1];
        LOAD_VEC(NAME, %remaining);
        %;
```

where

- `%count` is a lexical function whose expansion is the number of expansions of the containing macro that have so far completed.
- `%length` is a lexical function whose expansion is the number of actual arguments in the call.

Example Call:

```
INIT_LOCAL_VEC(TABLE2, 1, 5, 9-5);
```

which expands to

```
local
    TABLE2 : VECTOR[3];

TABLE2[0] = 1;
TABLE2[1] = 5;
TABLE2[2] = 9-5;
```

Figure 15: BLISS-11 Iterated Macros

- `exitcase`,
- `exitset` (exit one arm of a case for one value of the selector), and
- `exitselect`.

The optional level (whose enclosing brackets distinguished it from the optional expression) allowed specifying multiple levels of the given construct to be exited; the default was `[1]`. Finally, the optional *expression* specified the value to use for the exited construct, if one were needed because of its usage context. `return` was also regarded as an escape, although it did not allow a level specification.

The methodological difficulties of characterizing the "target" of an escape in terms of the kind of syntactic container and a level count are obvious: any change in the intermediate range may easily require a coordinated change in the escape, but the need for that change could be overlooked too easily.

In BLISS-11, labels were introduced into the syntax – virtually any construct could have a label. To avoid parsing ambiguities in dealing with undeclared names, labels had to be declared. Finally, the set of escape constructs were combined into a single `leave` construct written as:

`leave` *label-name*

or

`leave` *label-name* `with` *expression*

(As a pragmatic concession to popular usage, the single level `exitloop` with optional *expression* was also retained.)

Linkages and Environment Integration

The challenge of environment integration on the PDP-11 was quite different than that on the PDP-10: many options for the -10 did not apply and new options were needed for the PDP-11.

A key limitation of the options in BLISS-10 was that they applied to a complete compilation, and indeed, to a complete program. A more flexible design was needed that allowed properties to be specified on a routine by routine basis. This led to introduction of a new *linkage declaration.*

A linkage declaration establishes a name for a specific combination of routine calling conventions. Such a name can then be used as an attribute in subsequent routine declarations. Consider this example:

```
linkage
    NONSTD  = BLISS(register=1, register=2, stack);

routine LINKUP (FOREWARD, BACKWARD, HOME) : NONSTD =
    begin
    ...
    end;
```

In this example, the linkage name `NONSTD` is declared as a variation of the `BLISS` type of linkage in which the first two parameters are passed in registers R1 and R2, respectively, while the third (and any subsequent) parameter is passed on the stack. That linkage name is then used in the declaration of routine `LINKUP` to specify its linkage conventions.

Figure 16 summarizes the linkage types that were part of BLISS-11.

File Inclusion Declaration

The utility of collecting common, shared declarations of literals, macros, structures, and so on, into one or more files is obvious as soon as a program grows beyond what is convenient to manage (edit, list, compile) as a single file.

On the PDP-10, such shared files were "included" into a using source either through the concatenation of input files that was an integral part of the TOPS-10 command language (so that the compiler read the result of the logical concatenation as its input file) or through creation of temporary files and their subsequent compilation in a command script. Both practices had disadvantages. First, a compilation unit had to be split into at least two files (in addition to the files to be included): a header file that included the module header information (module name, certain option selections) and the main body of the module–this resulted in more pieces to manage. Second, the relationship between the two parts of a module and the files it included was separate from the files themselves–making it sometimes hard to keep pieces coordinated.

For BLISS-11, we solved both problems by introducing a declaration as in

```
require 'FILE.REQ';
```

`BLISS`	The default BLISS-11 linkage. Parameters are (normally) passed on the stack. There is no parameter count. A routine is called using `JSR PC,`*rtn* and returns using `RTS PC`. R0 is the return value register. Registers R1 through R5 are preserved by the callee, unless used to pass a parameter. Parameters on the stack are removed by the caller.
`FORTRAN`	The linkage conventions used by the several FORTRAN implementations on the PDP-11. The parameter list begins with a word aligned byte containing the parameter count and is formed in a separate block, whose address is passed in R5. (The parameter block can even be immediately inline following the call, in which case the count doubles as the displacement of a `BR`anch instruction that skips past the parameters.) A routine is called using `JSR R5,`*rtn* and returns using `RTS R5`. R0 is the return value register. Registers R1 through R5 are preserved by the callee.
`INTERRUPT`	The linkage conventions used by the hardware to initiate an interrupt service routine. There are two parameters on the stack: the program counter and processor status from the point of interruption. Such a routine is not (directly) called by software and returns using `RTI`. R0 through R5 must be preserved by the routine.
`EMT, IOT, TRAP`	The linkage conventions used to synchronously "call" an interrupt service routine, named after the specific hardware instructions that perform the call. They differ only in the different hardware vector addresses that are used to dispatch the interrupt; EMT includes an 8-bit code as the low byte of the instruction. Other parameters are passed on the stack by the caller and removed by the callee.

Figure 16: PDP-11 Linkage Types

This declaration causes the given file to be opened and its contained text used in place of itself. The effect is much like a C or C++ `#include` preprocessor directive, except that in BLISS it really is syntactically a declaration and can only be used in a place that a declaration is valid.

Other Miscellaneous Changes

There were other more minor changes, which we list in Figure 17 for completeness.

2.5 BLISS-11 Implementation and Use

Brender drafted the first version of a proposal for BLISS-11 in early 1972. Brender, Wulf and others at CMU debated and evolved the language proposal over the next several months.

Rather than adapt the BLISS-10 compiler framework, Wulf undertook a completely new design intended to allow more focus on optimization. (The small memory size available in the PDP-11's 16-bit address space made code quality especially critical.)

The first attempt at implementing the BLISS-11 compiler did not go well, in part because Wulf was personally focused on the design and construction of a Departmental research capability-oriented multi-processor based on the PDP-11. The Department very much wanted (needed!) BLISS-11 to use in implementing its operating system. In the summer of 1973, much of the BLISS-11 compiler work to that point was discarded and Wulf started over – this time becoming a major compiler implementer himself.

Meanwhile, DEC employee Paul Knueven relocated to Pittsburgh to provide full time liaison at CMU and to contribute to the implementation.

The second attempt worked out much better – so much better that a book[16] [46] and several Ph.D. theses [23, 25, 38, 36] in part either inspired or were inspired by that development. (Indeed, this design was so successful that it served as the basis for the next generation of BLISS compilers.)

[16] The "green book", as *The Design of An Optimizing Compiler* was more familiarly called, is still one of few monographs that provide a comprehensive description, both theory and pragmatics, of a complete highly optimizing compiler.

- Boolean expression semantics changed from unconditional complete evaluation to partial (short-circuit) evaluation (on the grounds that it allowed better performance).
- BLISS-10 allowed multiple selector expressions in the case and select constructs; this was limited to just one in BLISS-11 (on the grounds that there were no known uses of more than one).
- BLISS-10 allowed a variant of a routine called a function that supported Algol-like uplevel addressing; this was omitted from BLISS-11 (on the grounds that it was little used and efficient implementation on the PDP-11 was problematical).
- BLISS-10 included a simple co-routine construct; this was omitted from BLISS-11 (on the grounds that it was little used and efficient implementation on the PDP-11 was problematical).
- BLISS-10 allowed up to five 7-bit ASCII characters to be packed into a word, either right or left justified; BLISS-11 only allowed two 8-bit characters to be packed into a word (on the grounds this was the natural hardware representation).
- BLISS-10 allowed both = and ← as semantically equivalent assignment operators; BLISS-11 dropped the left arrow (on the grounds that it was no longer an ASCII character).
- BLISS-10 provided single precision add, subtract, multiply and divide floating point operators; BLISS-11 did not (on the grounds that floating point values could not be represented in the general purpose registers).
- BLISS-11 added a rotate (`ROT`) operator whose operation included the hardware carry bit (on the grounds that it provided access to the underlying hardware).
- BLISS-11 added an exception handling mechanism. An `enable` declaration provided handlers for its containing block and the `signal` *e* form of exit raised an exception "named" by the value of expression *e*.

Figure 17: Miscellaneous Minor BLISS-10 versus BLISS-11 Differences

Use at CMU

At CMU, BLISS-11 did become the implementation language for the Hydra operating system on the C.mmp multi-processor system.

Use at DEC

At DEC, Brender led the development project that made first use of BLISS-11 in a DEC product, PDP-11 FORTRAN IV-PLUS. This was a "state of the cookbook art" optimizing FORTRAN compiler and run-time library system for the PDP-11/45 with floating point processor. BLISS-11 was used as the predominant implementation language for the compiler; exceptions were made for interfaces to the host RSX-11M/D operating system and certain of the mathematically oriented service routines (the latter being shared with the run-time software). BLISS-11 was not used at all in the run-time because it was common practice at the time for customers to modify parts of the run-time software for special applications.

While it is notoriously difficult to reliably collect and interpret productivity data, data collected by the project suggested that productivity (debugged and delivered instructions per person day) on the BLISS-11 part of the compiler was 1.6 to 1.8 times that of the run-time system [4]. It was also clear that there were fewer bugs.

There were but a few other groups within PDP-11 Software Engineering that were willing to tolerate the use of a cross-compiler (BLISS-11 was hosted on the PDP-10) at a time when networks were just emerging (compiled code had to be hand carried from the compiling to the debugging systems). Nonetheless, Diagnostics Engineering did become a significant and enthusiastic user of BLISS-11.

Once again BLISS had been successfully used at DEC to produce important products that fulfilled their performance, quality and schedule goals.

Property	VAX-11
Word size	32 bits
Addressable unit	8-bit byte
Byte pointer	(just an) address
Byte access	fetch sign or zero extends (any size up to 32 bits)
Registers	16 (special purpose: AP, FP, SP, PC)
Floating point	used general registers (double precision used two) (quadruple precision used four)

Figure 18: Summary of VAX-11 Properties

3 The Second Generation of BLISSes: 1975-1978

During 1973 and 1974, Digital made several attempts to design extensions to the PDP-11 architecture that would eliminate the critical 16-bit limit on addressing–none of them were deemed attractive enough to proceed past the paper proposal stage. Finally, it was decided to initiate a new design whose most fundamental characteristics are: 32-bit address and word size, 8-bit addressable bytes, with PDP-11 compatible data types. (The summary of properties in Figure 18 can be compared with those in Figure 12.)

With a new hardware design starting up, there was considerable agreement (but not universal to be sure) that a high level language was needed to write the new software that would go with it. But which language?

The Great Language Debate

The great language debate raged for close to a year. There were evaluation committees and study reports, and enormous emotional energy spent supporting favorite candidates and criticizing others. Pascal and BLISS were the leading candidates, while PL/I, RTL-2 and a smattering of others also had advocates. (C had yet to emerge from Bell Laboratories, although we knew of it.)

Bill Segal remembers the resolution this way[33]:

> *...the first decision (by Portner and Bell) was Pascal and a team was actually designated and took residence on that floor. One to two weeks later, the team, their futons and all, disappeared. All I heard was that suddenly the decision has been changed in favor of BLISS. Ed Fauvre was the responsible manager under Portner. Bill Slack was the original manager for Development Methods. Ed asked me to take over [at that point].*

The Grand Plan

The decision to use BLISS as the implementation language came in the early Fall of 1975, BLISS-VAX (as it was initially called) would be based on BLISS-11 By October, a small BLISS development team of five programmers headed by Brender was formed to begin the language design and compiler development efforts.

Development Methods management put together the *BLISS PROGRAM PLAN* [34]. In its final form in April 1976, this 70 page plan, complete with schedules and budgets for the years 1976 through 1978, set forth the major milestones for new BLISS-VAX compilers (cross and native), ongoing BLISS-10 and BLISS-11 maintenance and enhancement, a supporting debugger, BLISS documentation, training for both programmers and field personnel, coding standards and conventions, program monitoring and metrics, eventual compiler productization, and risks and contingency plans. The Development Methods Department was expected to grow rapidly to 16 people and the total cost was projected at nearly $1.7 million, an extraordinary investment in *software tools* for DEC at that time[17].

Evolving Goals

The original vision for DEC's new generation of BLISSes had seemed simple enough:

- Extend BLISS-11 in as straightforward a manner as possible to make it "fit" the emerging new VAX design
- Clean up a few known language problem areas along the way
- Eventually adapt BLISS-10 to be more like BLISS-VAX

[17] DEC's revenue was $736 million in the fiscal year ending June 1976 [10] and its entire software development organization was just 328 programmers in February 1976 [34].

But three forces complicated the vision and our aspirations:

- Trying to converge the BLISS-10 and BLISS-11 variants of the language at the same time that we tried to extend it for VAX proved technically much harder than originally envisioned.
- Management concluded that the new BLISS must be a vehicle for portable programming. Having chosen to start with a language foundation that emphasized machine access and machine dependence, the portability goal forced even more radical change than expected.
- Once we had broken the bonds of even close compatibility with BLISS-11, it was very hard to limit the reach of our language redesign.

3.1 Language Evolution

In the following sections, aspects of the BLISS language evolution are grouped according to their motivation: hardware, portability, and other general improvements.

3.1.1 Hardware Matters

Most of the changes motivated directly by the VAX hardware were well served by the BLISS-11 example.

`BYTE`, `WORD`, and `LONG`

For compatibility with the PDP-11, VAX retained the term *word* to mean two bytes; it coined the term *longword* to mean four bytes. The longword sized value became the size of all values in BLISS-VAX, of course. The keyword `LONG` was adopted to indicate a value of this size.

Signed and Unsigned Fields

During the design of BLISS-11, we had waffled over whether subfields of a word should be signed and eventually stayed with unsigned, for compatibility with BLISS-10. But as experience continued to accumulate, it seemed clearer and clearer that there was a valid need for both.

Thus, the `SIGNED` and `UNSIGNED` *sign extension attributes* were introduced. The `<>` operation was extended to have an optional third parameter that specified whether to perform sign extension (1)

or zero extension (0) on a fetch; it was allowed but ignored for an assignment (which always did simple unchecked truncation). Thus, given

```
global
    A : BYTE,
    B : BYTE SIGNED,
    C : BYTE UNSIGNED,
    D;
```

the following pairs are equivalent:

```
D = .A;     and     D = .A<0, 8, 0>;
D = .B;     and     D = .B<0, 8, 1>;
D = .C;     and     D = .C<0, 8, 0>;
```

Floating Point

We could have formulated at least the single precision floating point operations as infix and unary operators on at least the PDP-10 and VAX, but we did not. For consistency, completeness and uniformity across all our platforms, these operations were formulated as special functions built into the compiler that took one or two address operands to specify the inputs and an additional address operand that specified where to store the result.

The resulting notation is not very convenient or intuitive for floating point intensive applications. But at least it makes the operations available for more limited purposes without needing to resort to assembler or another language. Moreover, because the operations are built in to the compiler, the optimizer phases can know exactly what data segments are read and written so that the quality of the compiled code need not be compromised as a consequence of the awkward notation.

Unknowingly, this approach neatly solved the problem for several future BLISSes as well.

3.1.2 Portability

Portability is often viewed as a property of a programming language. That is, it is expected that (almost) all programs written in that language will behave the same when compiled for and executed on diverse machine architectures. If the machine architectures are not too different, and

Characteristic	Name	PDP-10 Value	PDP-11 Value	VAX Value
Bits per BLISS value (word)	%BPVAL	36	16	32
Bits per Addressable Unit	%BPUNIT	36	8	8
Addressable units per BLISS value (%BPVAL/%UPVAL)	%UPVAL	1	2	4
Bits per address	%BPADDR	18	16	32

Figure 19: Predefined Literals for Portability

more importantly, if the main data types such as integer and floating point values have similar properties, this view is often true enough for many purposes.

But when the language is untyped, so that the fundamental (and only) data type is the machine sized word, then portability becomes much more a property of a program rather than the programming language as such. Portability needs to be planned in advance rather than merely tacked on as part of a porting exercise[18].

Our challenge was to provide the language mechanisms that would make it as straightforward as possible to write portable programs using BLISS.

Predefined Literals

Predefined literals were introduced whose values described characteristics of the target architecture of a compilation. These are shown in Figure 19.

Predefined Structures

The predefined literals made it possible to declare a set of standard structures as shown in Figures 20, 21, 22, and 23.

[18] This is not to say that data typing, of itself, is a panacea. Many programmers have learned the hard way, for example, that without careful forethought the C types `int`, `long` and `void *` provide enormous opportunity to compromise the portability of a program.

```
! Vector of signed or unsigned components
!
structure
    VECTOR[I; N, UNIT=%UPVAL, EXT=0] =
        [N*UNIT]
        (VECTOR + I*UNIT)<0, %BPUNIT*UNIT, EXT>;
```

where the formal names of the structure have the following meanings:

Formal Name	Meaning
I	The number of the element to be referenced (0 origin)
N	The number of elements in the vector
UNIT	The number of addressable units in each element (defaults to %UPVAL)
EXT	The sign-extension rule (0 means zero-extend, 1 means sign-extend)

Figure 20: Predefined Structures for Portability: VECTOR

```
! Bitvector: a vector of 1-bit components
!
structure
    BITVECTOR[I; N] =
        [(N + (%BPUNIT-1))/%BPUNIT]
        (BITVECTOR + I/%BPUNIT)<I mod %BPUNIT, 1, 0>;
```

where the formal names of the structure have the following meanings:

Formal Name	Meaning
I	The number of the element to be referenced (0 origin)
N	The number of elements in the vector

Figure 21: Predefined Structures for Portability: BITVECTOR

```
! Block
!
structure
    BLOCK[O, P, S, E; BS, UNIT=%UPVAL] =
        [BS*UNIT]
        (BLOCK + O*UNIT)<P, S, E>;
```

where the formal names of the structure have the following meanings:

Formal Name	Meaning
I	The number of the element to be referenced (0 origin)
O	The offset from the base of each block of a field
S	The size (in bits) of a field
E	The sign-extension rule (0 means zero-extend, 1 means sign-extend)
BS	The size (in UNITs) of each block
UNIT	The number of addressable units in each element (defaults to %UPVAL)

Figure 22: Predefined Structures for Portability: BLOCK

```
! Vector of Blocks
!
structure
    BLOCKVECTOR[I, O, P, S, E; N, BS, UNIT=%UPVAL] =
        [N*BS*UNIT]
        (BLOCKVECTOR + (I*BS + O)*UNIT)<P, S, E>;
```

where the formal names of the structure have the following meanings:

Formal Name	Meaning
I	The number of the element to be referenced (0 origin)
O	The offset from the base of each block of a field
S	The size (in bits) of a field
E	The sign-extension rule (0 means zero-extend, 1 means sign-extend)
N	The number of elements (blocks) in the vector
BS	The size (in UNITs) of each block
UNIT	The number of addressable units in each element (defaults to %UPVAL)

Figure 23: Predefined Structures for Portability: BLOCKVECTOR

In principle, it was not necessary to predefine these structures–they could have been defined by any user that needed them. However, we did build them into the language definition and implementation for several reasons.

First, we knew from several years experience using BLISS that these were the dominant structures that would be needed by most users most of the time.

Second, and actually more importantly, we were not prepared to build a debugger with support general enough to handle the full power of BLISS expressions. We took advantage of the fact that this particular set of just four structures was general enough for most purposes to build support for them directly into the debug symbol table representation and the debugger.

Third, and perhaps something of a rationalization, we believed that it was far more valuable for a program to include within itself "dumper" routines that could be called from a debugger to provide an interpretation of a data structure. Such an interpretation can be made much closer to the problem domain being addressed than any mere field by field data structure dump[19]. (An example of this is discussed in Section 4.2, see especially Figure 48.)

Address Comparisons

When we added the unsigned relational operators in BLISS-11 (see Section 2.4.1), we weren't thinking about transportability–and we got it wrong. What was needed was a set of operators specifically for comparing addresses that could be implemented on the PDP-10 using signed comparison instructions (since 18-bit addresses compare the same whether viewed as signed or unsigned integers) and implemented on the PDP-11 and VAX using unsigned comparison instructions. The unsigned relationals were still needed for those occasions when unsigned interpretation was intended regardless of the target environment (and the lack of unsigned comparison instructions on the PDP-10).

[19] We even brainstormed the notion of a BLISS attribute that would specify the dump routine to be called by a debugger for an examine command on a data segment, but we never followed through. Unfortunately there seems no way to get the same effect in other programming languages. In retrospect and hindsight, it appears that the ability to call a user routine–whether dump routine or otherwise–from debugger command level is good enough.

We added address versions of the relational operators, naming them after their BLISS-10 counterparts but with a suffix "A" appended. The result was three sets of relational operators for signed and unsigned integer and address comparisons, as in:

Comparison	Operators
Signed integer:	EQL, NEQ, LSS, LEQ, GTR, GEQ
Unsigned integer:	EQLU, NEQU, LSSU, LEQU, GTRU, GEQU
Address:	EQLA, NEQA, LSSA, LEQA, GTRA, GEQA

Character Handling Functions

Character (string) data on the PDP-10 was typically stored five 7-bit characters or six 6-bit characters in a 36-bit word. On the the PDP-11 and VAX it was stored one 8-bit character in an 8-bit byte. To make portable character handling feasible across these domains, we developed a set of *Character Handling Functions*, as shown in Figure 24.

In most respects, the character handling functions are quite straightforward and not particularly surprising. The trick, if you will, was to formulate them to be abstract enough to be compiled to multiple machine architectures generally as well as efficient on the machines we particularly cared about.

One aspect of the PDP-10 does show through in a way that might not be expected. Notice that the `CH$A_RCHAR` and `CH$A_WCHAR` functions are defined to advance the character pointer *first* and then to read/write the character at the updated position, rather than the opposite order. This choice directly reflects the PDP-10 hardware `ILDB` (increment and load byte) and `IDPB` (increment and deposit byte) instructions, which performed the increment and access in that order.

Lexical Conditionals

Conditional compilation, the conditional inclusion or exclusion of program text based on a compile-time test, was a well established feature of many assembly languages but rarely is available in high level languages, even today.

In BLISS-10, very limited kinds of conditionalization could be achieved based on the conventional `if-then-else` form, exploiting the fact that a test whose value was known at compile time could (and was) used to eliminate one alternative or the other. But this could only be used in the executable part of a program and was not applicable to declarative parts generally.

Function Name	Interpretation
CH$ALLOCATION(*n*, *cs*)	Allocation size required for a character sequence of *n* characters each of size *cs* bits (normally used as the size in a VECTOR structure attribute)
CH$PTR(*addr*, *i*, *cs*)	Form the character pointer that refers to the *i*'th element of the character sequence which begins at address *addr* where each character is of size *cs* bits.
CH$PTR(*ptr*, *i*)	Given character pointer *ptr*, form the character pointer for the character position that is *i* characters later.
CH$RCHAR(*ptr*)	Given character pointer *ptr*, fetch the character stored at that location
CH$A_RCHAR(*addr*)	Given the address of a location that contains a character pointer, advance and update the character pointer to refer to the next character position and then fetch the character stored there
CH$WCHAR(*c*, *ptr*)	Given character pointer *ptr*, store the character *c* at that location
CH$A_WCHAR(*c*, *addr*)	Given the address of a location that contains a character pointer, advance and update the character pointer to refer to the next character position and then store the character *c* at that location
CH$MOVE(*n*, *sptr*, *dptr*)	Given a source and destination character pointers *sptr* and *dptr*, respectively, copy *n* characters from the source to the destination
CH$FILL(*c*, *dn*, *dptr*)	Fill *n* character positions starting at the destination character position *dptr* with the character *c*
CH$*xxx*(*n*1, *ptr*1, *n*2, *ptr*2, *fill*)	Compare the *n*1 characters at character position *ptr*1 to the *n*2 characters at character position *ptr*2, filling with *fill* as needed, for

CH$*xxx*	Relation
CH$EQL	Equality
CH$NEQ	Inequality
CH$LSS	Less than
CH$LEQ	Less than or equal
CH$GTR	Greater than
CH$GEQ	Greater than or equal

Figure 24: Portable Character Handling Functions

```
%if lexical-test        ! Any constant expression
%then lexemes...        ! The "then-part"
%else lexemes...        ! The "else-part" (optional)
%fi
```

Figure 25: Lexical Conditionals

BLISS-11 added a form of macro with conditional expansion (see Figure 14) which at least was usable in all parts of a module (declarative as well as executable). But, experience showed it to be generally awkward and clumsy to use.

We debated many months over whether to add a more direct form of conditional processing that, like macros, would take place as part of lexical processing and prior to syntactic parsing. Wulf recommended against it. There was concern that interactions between the conditional mechanism and macros generally would be complicated and hard to understand.

Eventually, a conditional compilation mechanism was adopted as shown in Figure 25. The lexical test could be any expression whose value was known at compile time. If the value was `true`, then the lexemes following `%then` up to the following `%else`, if present, or `%fi` otherwise, are included as part of the program, and those following the `%else` were ignored; otherwise, conversely.

Notice that `%if` required an explicit closing `%fi`, which was not analogous with the normal executable `if-then-else`–we simply could not find an attractive way to do without it!

We limited the interactions between this conditional mechanism and macros generally with a set of well-formedness rules regarding nesting as follows:

- If a macro body contains the lexeme `%if`, then it must also contain the matching `%then`, `%else` (if present) and `%fi`. This rule must be satisfied by the source file before *any* lexical processing has been performed.
- A similar restriction applies to an actual parameter of a macro call or lexical function generally.
- If a lexical conditional is nested in the then (or else) part of an outer lexical conditional, then a similar restriction applies (so that it is completely nested).

(Similar restrictions applied in other cases, but are beyond the scope of this discussion.)

We did pay heed to the cautionary advice to some extent, however. It was tempting to include an `%elsif` alternative that would allow a sequence of tests without using nesting, but we did not. We were also tempted to add a lexical form of the `case` selection construct, but we did not.

In retrospect, the lexical conditional mechanism accomplished everything that we wanted of it and has not been a source of abuse or confusion; in particular, the well-formedness restrictions have proved just right in this regard[20].

Namespace Management

As we evolved and expanded the language, we became concerned that the growing number of names would become too large to be intellectually manageable. We ended up with over 500 defined names in the BLISS language manual!

We certainly did not want to simply reserve all defined names across all variants of BLISS as portability goals might suggest. We also wanted some flexibility to introduce additional new capabilities over time without risking conflicts with existing programs if at all possible.

We divided the set of names into several categories as follows:

- A *reserved keyword* (such as `if` or `local`) has a fixed meaning and use; it cannot be declared as a name under any circumstances.

  The lexical operation names (such as `%if` or `%count`) technically all count as reserved keywords. But since no user names beginning with `%` could be declared in any case, they are not a potential source of conflict.

- An *unreserved keyword* (such as the `CALL` linkage type, or the names of module or program section options) can be declared as a user name. Unreserved names occur in the language definition for names that occur in very specific contexts and typically select among a fixed set of alternatives of some kind; that is, where a declaration or lack thereof of the name has no relevance.

- A *predeclared* name (such as the absolute value function `ABS`) can be used as an explicitly declared name. Such a declaration does make the predeclared meaning unavailable within the scope of the explicit declaration. (Conceptually, predeclarations exist in an imaginary scope

[20] Had we also included `%elsif` and some form of `%case`–purely syntactic sugar, really–these almost surely would have proven convenient and safe as well.

Name category	Count
Reserved keyword	
syntactic	107
lexical	70
Unreserved keyword	102
Predeclared	55
Built-in	
shared	19
BLISS-16	36
BLISS-32	103
BLISS-36	45

Figure 26: Predefined BLISS Names by Category

that contains the program being compiled and such names are subject to nested redeclaration in the normal way.)

- A *built-in name* must always appear in an explicit declaration. If it is declared by a built-in declaration, as in

```
builtin
    R0,     ! A hardware register name
    ARGPTR; ! A VAX linkage related function
```

then it has its predefined meaning; otherwise, it has the meaning given by its explicit declaration, just as if it were not a predefined name. Most of the predefined names in this category are functions that correspond to particular machine instructions of the target architecture whose effect cannot otherwise be achieved in a natural or direct manner.

The number of BLISS names in each category is shown in Figure 26. While 107 reserved keywords is rather large for a high-level language, it is a whole lot better than over 500!

Documentation

A single integrated BLISS Language Manual was developed to specify all three BLISS dialects simultaneously. The core language that was shared by all dialects became known as "Common

BLISS". Each feature, option or characteristic that was not common was clearly marked with the subset of dialect(s) of which it was part.

A variety of talks and papers (for example, [28, 17]) were developed to help people design portability into their programs where desired.

Portability Checking

To aid in making a program transportable, a `LANGUAGE` switch option was provided (used either as a compiler command switch or in a `SWITCHES` declaration in the source) which expressed the intention to compile the program with a particular set of BLISS compilers. The parameter could be either `COMMON`, which specified all BLISS dialects (which theoretically could–and later did!–change over time), or a list of specific dialects from among `BLISS16`, `BLISS32`, and `BLISS36`.

For example, if you intended your program to be used on either the PDP-11 or VAX, you would specify `LANGUAGE(BLISS16, BLISS32)`. In this context, the BLISS-32 compiler would give a warning if the attribute `LONG` was used, because that attribute was not available in BLISS-16.

Portable Declaration of Structures

Portable declaration of structures surely deserves mention in this section, but the means to do so are actually built using new language features that were introduced primarily for reasons other than portability as such. We will return to this topic in the next section after introducing field declarations and compile time processing.

3.1.3 General Improvements

Many of the general language improvements were motivated by weaknesses and limitations experienced while using BLISS-10 and BLISS-11, while others were motivated by a desire to actually increase the degree of hardware integration for each respective architecture. (Our management occasionally needed to be reminded that the latter was a reasonable and proper part of our goal set.)

Toward a More Extensible Syntactic Style

The BLISS-10 and BLISS-11 syntax for data segments allowed a structure name to appear followed by one or more data names and associated structure actual parameters. For example:

```
own
    VECTOR   A[10], B[2];
```

Further, for BLISS-11, a `BYTE` (or `WORD`) attribute could occur in front of the storage class keyword, as in

```
BYTE own
    VECTOR A[10], B[2];
```

As more and more attributes were introduced it became more and more problematical how to integrate them into the syntax in a way that would not appear ambiguous or be hard to read. Which should be allowed or required in front? Which should or must come later? Was the order important? Could some or all occur in either place? To resolve these questions, we restructured the syntax as follows.

First, we retained the "top most" division used by BLISS-10 and BLISS-11 for their traditional meanings: `own`, `global` and `external` for static storage, and `local`, `stacklocal` and `register` for automatic storage. Then follows one or more item declarations, which are separated by comma, with the last terminated by a semicolon. Each item consists of the declared name optionally followed by colon and the sequence of applicable attributes (in any convenient order).

Accordingly, the above example is written:

```
own
    A : VECTOR[10, BYTE],
    B : VECTOR[2,  BYTE];
```

We deliberately did not seek a syntax that would allow multiple items to share the same (complete) set of attributes. We felt that laying out the items individually, one per line, and generally in alphabetical order was much preferred as a coding style. We have never regretted that decision.

```
routine DOIT (PARAM_VEC : VECTOR) = ! "Desired" notation
    begin

    local
        LOCAL_VEC   : VECTOR[10];   ! Declares and allocates

    LOCAL_VEC[0] = .PARAM_VEC[0];   ! Problem!!!
    ...
    end;

own STATIC_VEC : VECTOR[10];        ! Declares and allocates

routine CALLER =
    begin

    DOIT(STATIC_VEC);               ! Address of STATIC_VEC

    end;
```

Figure 27: Level of Indirection Problem with Structures

The ref Structure Modifier

A lingering problem with structure usage was the inability to use the same structure for a parameter that was "passed by address" as for a local variable. Consider the example in Figure 27.

If you work through the definition and expansion of VECTOR introduced in Section 2.2.3, you discover that there is a level of indirection missing in the access of the parameter PARAM_VEC. This corresponds to the fact that PARAM_VEC is not itself a vector but rather a data segment whose contents is the address of a vector.

The most common resolution up to this time was to have a second structure declaration, say, called REF_VECTOR in this case, that was identical to VECTOR except that there was an additional dot operator applied to the structure name. For example:

```
structure
    REF_VECTOR[I; N] =
        [4]
        REF_VECTOR + I - 1; ! 1-origin referenced vector
```

Also note that the allocation expression is not N but a constant size appropriate for holding an address (4 on VAX). Writing and maintaining such near duplicate pairs of declarations was tedious and error prone.

It was also possible to use a bind declaration to avoid introducing a second structure name, but at the cost of introducing a second name for the parameter itself:

```
routine DOIT (PARAM) =
    begin

    bind
        PARAM_VEC = .PARAM : VECTOR;

    ...
```

This approach was not very attractive or popular either.

We resolved this problem by introducing a keyword ref, which could be used to prefix the structure name in a data declaration. The routine declaration could then be written

```
routine DOIT (PARAM_VEC : ref VECTOR) = ...
```

When PARAM_VEC is accessed, the presence of ref on its declaration causes an extra fetch operation to be applied to the structure base.

Interestingly, we did not define `ref` so that it could be used in combination with the `BYTE`, `WORD` or `LONG` allocation keywords; it would not have been hard to do so. But it turned out that the following structure declaration provided a perfectly good substitute:

```
structure SCALAR[ ; size=%UPVAL, ext=1] =
    [size]
    scalar< 0, size*%BPUNIT, ext>;
```

This allowed a routine to be declared that took the address of a `BYTE`, for example, and accessed that byte much like a structure, as in:

```
routine GETB(PB : ref scalar[BYTE]) =          ! Get byte at .PB
    .PB[];

routine SETB(PB : ref scalar[BYTE], VAL) =     ! Set byte at .PB
    PB[] = .VAL;
```

Besides providing the desired functionality, it was actually cleaner in that it avoided special case interpretation of the `BYTE`, `WORD` and `LONG` keywords as though they were an implicit structure name.

Fields and Structure Accessing

The use of macros to provide names for fields of record-like structures (as illustrated in Figures 5 and 13) has two serious limitations:

- There is nothing that keeps the macro name for one kind of record from being mistakenly used to access a data segment of some other kind of record.
- The allocation of fields within a record is necessarily done by hand. This is not convenient in the first place, is error prone from a maintenance perspective in the long run, and is certainly not portable across architectures as diverse as DEC's PDP-10, PDP-11 and VAX.

The first problem was addressed by introducing the new field declaration and related attribute as described next. The second problem was addressed not by new language mechanisms, but through

application of new general purpose compile time language mechanisms; these are described later in this section.

The field declaration and its use is illustrated in Figure 28, which should be compared with the example in Figure 5.

In this example, the first part of the field declaration defines a name (`FORM1`) for a set of field names, each of which have the given numeric "expansions". Notice that these field names correspond directly with the macros that were defined in the earlier Figure. The second part of the declaration defines one additional field name (`Word1Flag2`) which is used separately.

The declaration of `DATUM` in this example adds a *field attribute*, which specifies the field names, or sets of field names, that are allowed in structure accesses involving that data segment. In particular, this makes the immediately following structure accesses, which are the same in both Figures, valid. Note that since `Word1Flag2` is not included in the field attribute for `DATUM`, the structure access `DATUM[Word1Flag2]` is not allowed (and results in an error diagnostic).

This Figure goes on to illustrate what in C++ terms would be considered an object of a derived type. The data segment `DATUM2` is just like `DATUM` except that the field `Word1Flag2` is valid when accessing it.

A field name does not necessarily have to provide all the arguments to a structure access (but can't supply too many). In particular, in an example involving a vector of blocks, the first argument would typically be an expression that indexes into the appropriate element of the array while the second argument would typically be one of the same field names that would be used with a single instance of that block.

Psects and Alignment

BLISS-10 and BLISS-11 had rather limited models of program storage. There was:

- Read-only code
- Read-only data (constants, literals)
- Read-write data (data segments, the stack)

For VAX, the software environment was planned to include a rather general notion of named program sections, each of which could be given a variety of attributes (read vs read-write, absolute

```
structure
    REC[O, P, S; N] =
        [N]
        (REC + O)<P, S>;

field
    FORM1 =
        set
        Word1LowHalf = [0,  0, 18],
        Word1Flag1   = [0, 18,  1],
        Word2Full    = [1,  0, 36]
        tes,
    Word1Flag2       = [0, 19,  1];

own
    DATUM : REC[2] field(FORM1);

DATUM[Word1LowHalf] = -1;
DATUM[Word1Flag1]   =  0;
DATUM[Word2Full]    =  1;

! Example of a "derived" object
!
own
    DATUM2 : REC[2] field(FORM1, Word1Flag2);

DATUM2[Word1Flag2] = .DATUM2[Word1Flag1];
```

Figure 28: BLISS Field Declaration and Attribute Example

```
own
    A,
    B;

psect OWN = BETA(nowrite);
own
    C;

psect OWN = DELTA(execute, align(8));
own
    D : VECTOR[100] align(4),
    E : psect(BETA);
```

The data segments for data segments A and B are allocated in the default program section for the storage class OWN (static storage). Data segment C is allocated in program section BETA, as is data segment E; BETA is not writeable. Data segment D is allocated in program section DELTA, which can be executed.

Figure 29: Psect and Alignment Examples

vs relocatable, required alignment, concatenated vs overlaid with other contributions of the same name, and so on).

To support this aspect of the VAX environment, we introduced a new *psect declaration* as well as allowed psect names to appear as attributes of an individual data segment.

While the VAX generally did not require data to be aligned, performance was better if it was. So we introduced a new *alignment attribute* which could appear on the declaration of a data segment.

Figure 29 illustrates these ideas.

Linkage Enhancements

BLISS-10 offered limited flexibility in the choice of register usage in a program–limited in that the same choice had to be used throughout that program (Section 2.2.8). BLISS-11 improved on this by introducing linkage declarations to name a particular combination of calling convention properties and providing some control over where parameters were passed (Section 2.4.2).

Common BLISS enhanced linkages further by adding:

- Control over which registers are preserved, not preserved or used at all by a routine
- Output parameter registers that allowed multiple return values
- Global register data segments shared across a set of routines.

The examples in Figures 30, 31 and 32 are used to illustrate these capabilities. The details of the linkage declaration vary from architecture to architecture, but all share the same style. There are also a variety of mostly straightforward rules regarding the use of registers in more than one of the manners described. All of these details are beyond the scope of this discussion. For concreteness, these examples all use BLISS-32 for VAX.

Control over register usage is straightforward as illustrated by the declaration and use of linkage `LINK1` in Figure 30. Each register had a default usage based on the linkage type, `CALL` in this example. This linkage is defined to use the VAX `CALLS` instruction. The first parameter is to be passed in register 0, which also happens to be used to return the result (which could not be changed on VAX). Register 1 is specified as `nopreserve`, meaning that it could be used by the called routine and need not be restored. Registers 2 and 3 are specified to be `preserved`, meaning that they can be used by the calling routine in which case they must be saved on entry or restored on exit; the caller can leave results needed later across the call in these registers. Finally, registers 4 though 11 are specified to be `notused`, meaning they must not be used in any way by the code for routine `RTN1`; the caller can treat these registers the same as preserved. In this example, routine RTN1 could contain a call to another routine that had the same linkage but it could not call a routine that had a standard linkage because the `notused` property has to be transitive to include any called routines.

A common paradigm in assembly language programming is to call a routine that looks up or computes characteristics about an entity and returns several pieces of information at once using the general registers. In cases like the computation of mathematical *sin* and *cos* it is cheap to compute both values and return both at once.

In Figure 31 the semicolon in the parameter-like list following the linkage type ends specification of input parameters and begins specification of output parameters. Like an input parameter, an output parameter is essentially a local variable of the called routine. However, while an input parameter is initialized with the value of an actual parameter at the time of a call, an output parameter is not initialized at the call at all. It must be assigned within the called routine (which assigns to the specified register). The contents of the output registers are then assigned following

```
linkage
    LINK1   = CALL(register = 0)
            : nopreserve(1)
              preserve(2, 3)
              notused(4, 5, 6, 7, 8, 9, 10, 11);

routine RTN1 (PARAM1) : LINK1 =
    begin
    ...
    end;
```

Figure 30: BLISS Linkage Register Usage Example

the call; the actual parameters that correspond to output parameters are much like the left-hand side of assignments that occur following the call. A semicolon also separates the input parameters and the output parameters at the call site. In the example, values are assigned to local variables `X`, `Y` and `Z` by the call to `RTN2`. Notice that `Z` is a byte rather than a longword; this is not relevant to `RTN2` and is handled by the normal assignment semantics in the caller.

Another common paradigm in assembly language programming is to structure a subsystem so that there are a small number of entry points and once within the subsystem some of the registers are used as shared global storage. In some cases, a similar effect can be achieved with register input parameters but this fails to allow updating the registers and having the results reflected in a calling routine. We made this paradigm available in BLISS using global register data segments.

There are three parts to the global register mechanism, as illustrated in Figure 32.

The global register declaration introduces a new local data segment with the given name allocated in the given register; in this example data segment `REG` is allocated in register 3. If a call occurs in the scope of the declaration, the linkage of the called routine must either indicate that that register is preserved (or not used), in which case the register is unrelated to any action in the called routine, or must specify that that register is a global register with the same name, in which case the register lifetime extends dynamically into the called routine.

An external register declaration is only valid within a routine if the linkage for that routine specifies a matching register and name, in which case the global data segment can be manipulated within

```
linkage
    OUTPUT3  = CALL(; register = 2, register = 3);

routine RTN2 (OUTLOC1, OUTLOC2) : OUTPUT3 =
    begin
    ...
    OUTLOC1 = 1;
    OUTLOC2 = 2;
    return 0;
    end;

routine LOOKUP =
    begin

    local
        X   : LONG,
        Y   : WORD,
        Z   : BYTE;

    X = RTN2(; Y, Z);

    end;
```

Figure 31: BLISS Linkage Output Register Example

```
linkage
    GLB_REG  = CALL : global(REG = 3);

routine ENTRY () =
    begin

    global register
        REG = 3;

    REG = ...;
    INNER();

    ! Possibly updated REG available

    end;

routine INNER : GBL_REG =
    begin

    external register
        REG = 3;

    ... use and/or update REG...

    end;
```

Figure 32: BLISS Linkage Global Register Example

that routine and no other allocation of that register can occur, either explicitly or implicitly by the compiler. If no external register declaration corresponds to the global option in the containing routine linkage, then the global register can be preserved by the calling routine.

The linkage declaration acts like a bridge that ties corresponding global and external linkage declarations together in a way that can be statically checked at compile time for consistency.

To the best of my knowledge, the Common BLISS global data segment mechanism is unique among high level programming languages.

Compile-time Processing

We introduced a considerable number of lexical functions, as illustrated in Figures 33 and 34. That Figures are not intended to be definitive in any sense, rather they seek to suggest the scope and variety of compile-time lexical facilities that were provided.

Of particular interest, however, is use of the `%assign` lexical function in combination with compile time variables. The declaration

```
compiletime
    Y = 0;
```

declares a "literal" (which must be given an initial value) whose value can be changed at compile time using `%assign`. The value of the literal is the value most recently assigned as in

```
%assign(Y, Y+1)
```

which increments the value of Y. Lexically the `%assign` function disappears; it is used solely for its side-effect of changing the value of a compiletime variable.

The `%quote` lexical function is also of special interest. BLISS-11 had introduced the undeclare declaration, which logically terminated the scope of a prior declaration. It was originally intended as a way to prevent accidental access to data segment names defined in outer blocks, for example, when that data segment was being manipulated using an alias with a different structure. But it also made the name available to be declared again.

Function	Expansion
%string('ABC',23, %B'-1111',,phi)	'ABC23-15PHI'
%exactstring(4,'12345678')	'1234'
%char(65)	'5'
%charcount('ABC',234)	6
%explode('ABC',23)	'A', 'B', 'C', '2', '3'
%remove(Q) [where Q is (A+1)]	A+1 *remove outer parens*
%name('+302',beta)	+302BETA *unusual spelling*
%quotename('FOO','BAR')	FOOBAR *form name, don't expand*
%null(0)	0 *not a null sequence*
%identical(ABC 5,ABC %B'101')	1 *sequences are identical*
%isstring(BETA, 'BETA')	0 *first BETA not a string*
%ctce(ALPHA[1])	0 *not a compile-time constant expression*
%ltce(.ALPHA[1])	0 *not a link-time constant expression*
%nbitsu(7, 2)	3 *number of bits as unsigned values*
%nbits(7, 2)	4 *number of bits as signed values*
%allocation(X) [X a default scalar]	%UPVAL
%size(VECTOR[10,WORD])	20 *BLISS-16/32 only*
%fieldexpand(DCB_E)	1,0,%BPVAL,0
%assign(X,2+3) [X is compiletime]	ϕ *associates 5 with X*
%number(Y) [Y declared literal 6]	6
%declared(A)	1 *A is declared*
%switches(OPTIMIZE, NOCODE)	1 *these switches are on*
%bliss(BLISS32)	1 *for BLISS-32 compiler*

Note: ϕ indicates no lexemes (an empty expansion)

Figure 33: Summary of Lexical Functions–Part 1

Function	Expansion
%error('Error text')	*ϕ steps error count, emits compile time error message*
%warn('Warning text')	*ϕ steps warning count, emits compile-time warning message*
%inform('Informational text')	*ϕ emits compile time informational message*
%print('Listing text')	*ϕ include text in listing*
%message('Terminal text')	*ϕ output text to terminal*
%errormacro('Error text')	*ϕ abort all macro expansions*
%title 'Listing title'	*ϕ specify listing title line*
%sbttl 'Listing subtitle'	*ϕ specify listing second line*
%quote lexeme, comma, or percent	*ϕ include in macro body*
%unquote (binds following name)	*ϕ force name binding*
%expand (binds and expands)	*ϕ force macro expansion*
%remaining	*unmatched actual parameters*
%length	*number of actual parameters*
%exititeration	*ϕ abort iteration of macro*
%exitmacro	*ϕ abort innermost macro*
%require('ERRMSG')	*include specified file*
%variant	*number from command line option*

Note: ϕ indicates no lexemes (an empty expansion)

Figure 34: Summary of Lexical Functions–Part 2

```
macro
    FANCY = CALL_FOO(2) %;

! At this point FANCY expands to CALL_FOO(2)

undeclare
    %quote FANCY;   ! End the current declaration of FANCY

macro
    FANCY   = CALL_BAR(3) %;

! At this point FANCY expands to CALL_BAR(3)
```

Figure 35: BLISS Macro Re-Declaration Example

In BLISS-11 there was no way to redeclare a macro name because the macro call was expanded before the undeclare declaration was syntactically parsed. This was surmounted by the new %quote lexical function which prevented macro expansion. This is illustrated in Figure 35.

The combination of macro and undeclare declarations facilitated by %quote meant that macros could be used at compile-time to hold varying lexeme streams.

The combination of

- A highly flexible macro mechanism generally, which in particular provides certain forms of iteration
- A flexible lexical condition facility (%if) to provide compile-time decision making
- compiletime declarations and %assign to provide compile-time numeric manipulation
- macro and undeclare declarations together with %quote to provide compile time lexeme stream manipulation
- A rich set of lexical functions more generally

together provide truly extraordinary compile time power.

Later sections of this paper illustrate some of the ways in which this power came to be exploited[21].

Portable Declaration of Structures

One of the first applications of the BLISS compile time mechanisms was to establish a set of macros to aid in defining portable data structures. There have been a number of variations in this set over the years, but this sketch will illustrate the basic idea. To begin, consider Figure 36 and compare it with Figures 5 and 28.

The `$structure`[22] macro takes one argument, an identifier that will later serve as the name of the "record type" being declared. This macro call

- Initializes various compiletime names that are used to keep track of the allocation as it progresses
- Remembers the identifier for later use (in the body of a macro it declares)
- Emits (expands to) the beginning of a field declaration. Note that the fieldset name to be declared is formed from the parameter name concatenated with the suffix `__FIELDS`.

Following the `$structure` macro call are a series of normal field name definitions, except that instead of a series of numbers there is a macro call that mimics the name of a type. Typical macros include

`$address`	For an address
`$int`	For a signed integer (default size)
`$uint`	For an unsigned integer (default size)
`$bit`	For a bitfield or flag
`$bits(`*n*`)`	For a field of *n* bits

Each of these macros performs several computations to make an appropriate allocation: the allocation (number of addressable units and bit position within unit) is rounded up as appropriate to the alignment of the new component, the appropriate offset, position, size and sign extension values are emitted for the component, the largest alignment encountered is computed for later use

[21] For a more lighthearted example, see [32].

[22] By convention, names beginning with dollar sign or containing double underscore are reserved for use by the BLISS extensions package.

```
$structure(FORM1)

    Word1LowHalf = [$address],
    Word1Flag1   = [$bit],
    Word2Full    = [$int]

$end_structure

! which expands (in BLISS-36) to
!
!field
!   FORM1__FIELDS =
!       set
!       FORM1__BASE  = [0, 0,  0, 0],
!       Word1LowHalf = [0, 0, 18, 0],
!       Word1Flag1   = [0, 18, 1, 0],
!       Word2Full    = [1, 0, 36, 1]
!       tes;
!
!literal
!    FORM1__SIZE = 2;
!
!macro
!    FORM1 = BLOCK[FORM1__SIZE] field(FORM1__FIELDS) align(0) %;

own
    DATUM : FORM1;

DATUM[Word1LowHalf] = -1;
DATUM[Word1Flag1]   =  0;
DATUM[Word2Full]    =  1;
```

Figure 36: BLISS Portable Data Structure Example

as the alignment of the record as a whole, and the allocation is again increased by the size of the component.

The `$end_structure` macro call completes the field declaration, after adjusting the size for any applicable alignment requirements. It then emits a literal declaration for the size of the record. Finally it emits a macro declaration that defines the original parameter name to expand to a `BLOCK` builtin structure attribute of the appropriate size followed by appropriate field and alignment attributes.

As a result of all this, the original parameter name becomes suitable to use as a sort of pseudo-type name in any kind of data segment declaration. Note that the `ref` structure attribute modifier can be used with this name in the obvious way.

In more sophisticated versions of these macros, there are means for handling unions and nested structures, reverse (negative offset) allocation from the base of the structure, vectors of and pointers to other pseudo-types, and so on.

In many cases, these macros could share implementation details based on the builtin literals such as `%BPUNIT`; in other cases, they were different for each respective target architecture. What mattered most, of course, was that the code that used these macros was target independent and adapted automatically to the target for which it was being compiled.

Precompiled Libraries of Declarations

Early in the development of the VMS operating system for VAX, we learned that large scale use of required files could severely compromise compilation time. Even in a reasonably small module, the time and system resources needed to process numerous required files, *most of whose declarations would not be used*, could easily dominate the overall compilation cost.

To remedy this, we introduced a means to precompile required files into a binary format that would facilitate subsequent incorporation into later compilations. The contents of so-called library files was limited to just declarations, in particular, those declarations that did not involve producing an object file (that is, no code or data allocation). Lexical conditionals were evaluated at precompile time, but otherwise the precompilation and subsequent library inclusion produced the same effect as direct source inclusion using a `require` declaration would have had.

Library files were included in a compilation using a new `library` declaration, as in:

```
library 'SYS$LIBRARY:XPORT';
```

The binary file format was designed so that inclusion of a library file involved only the reading of a name table to maintain a complete and consistent name space. All other aspects of a declaration were looked up and read from the library file only if and after the declared name was actually used elsewhere in the compilation.

3.2 Building the Compilers

The extensive(!) evolution of the language definition relative to BLISS-10 and BLISS-11 greatly complicated the task of first implementing compatible compilers and then bootstrapping them into the revised language. But we found a way.

3.2.1 BLISS-32C

The BLISS-11 compiler was the basis for producing a BLISS-32 cross-compiler on the PDP-10. Basically, the back-end or code generation part of BLISS-11 was completely replaced by a code generator for VAX-11, while the front-end was retained and modified to accommodate the new language requirements. However, the basic design and organization of BLISS-11 was carried over with little change.

The ultimate goal of bootstrapping BLISS-32 onto VAX-11 required some very significant changes in internal data representation. After a most difficult consideration of imprecise cost estimates and non-quantifiable risks, we decided to make the change in representation early. (In retrospect, this was definitely the better choice. The early experience designing for transportability proved very valuable in formulating some aspects of the new BLISS language; later on this experience would have been too late to have this benefit.) When the representation change was accomplished, the main compiler implementation task got under way.

Our environment was far from stable. Our own desire to improve BLISS led us to evolve and change the language while the option was still open to us. The VAX-11 instruction set specifications were still changing. Even the addressing mode structure changed significantly as late as April 1976, after we had demonstrated executable code on a hardware simulator. Fortunately, we had excellent working relationships with the VAX-11 design team and shared with them a mutual goal that BLISS and VAX-11 "fit" well together.

The first version of the BLISS-32 cross-compiler was released to the VAX-11 software developers in October 1976, just one year after the BLISS program began.

3.2.2 BLISS-36C

A compiler for the BLISS-36 dialect was needed for two reasons. First, we needed it to use in the bootstrap of BLISS-32 onto VAX-11 as described later. Second, since BLISS-36 was to replace the use of BLISS-10 in the long run, it was desirable to have a BLISS-36 processor as soon as possible to enable other development groups in the company to plan their conversion when it best fit their schedules. Neither of these motives, however, required support of the full BLISS-36 language or that the compiler provide the highest quality optimization – at least not initially.

Our original idea was to directly modify the BLISS-10 compiler to produce an interim compiler for BLISS-36. Several considerations, however, made this strategy increasingly less satisfactory. First, the revised BLISS language was evolving to be more different from BLISS-10 and -11 than we had originally foreseen. Second, the BLISS-10 compiler, originating in 1969, had already been pushed well beyond its original design objectives in both language features and optimization quality. Third, the new BLISS language incorporated and extended the macro facility of BLISS-11, which was quite different from the macro facility of BLISS-10. While the two macro mechanisms give equivalent effects in simple cases, established practice exploited both macro systems extensively in ways that could not be duplicated in the other.

The strategy that avoided these problems was to build a source to source translator to convert from BLISS-36 to BLISS-10 for use as a preprocessor to the BLISS-10 compiler. The front-end of the translator was built by adapting the front-end of the BLISS-32 compiler. The main new code consisted of a "back translator" from the internal symbol table and tree representation produced by the front-end to source notation.

Figure 37 illustrates the manner in which the BLISS-32 and BLISS-36C compilers shared common code for the front-end while having quite different subsequent parts. This strategy assured strict compatibility between the two compilers. For example, since macro processing was completely carried out in the front-end, the BLISS-10 intermediate representation did not contain any macros at all.

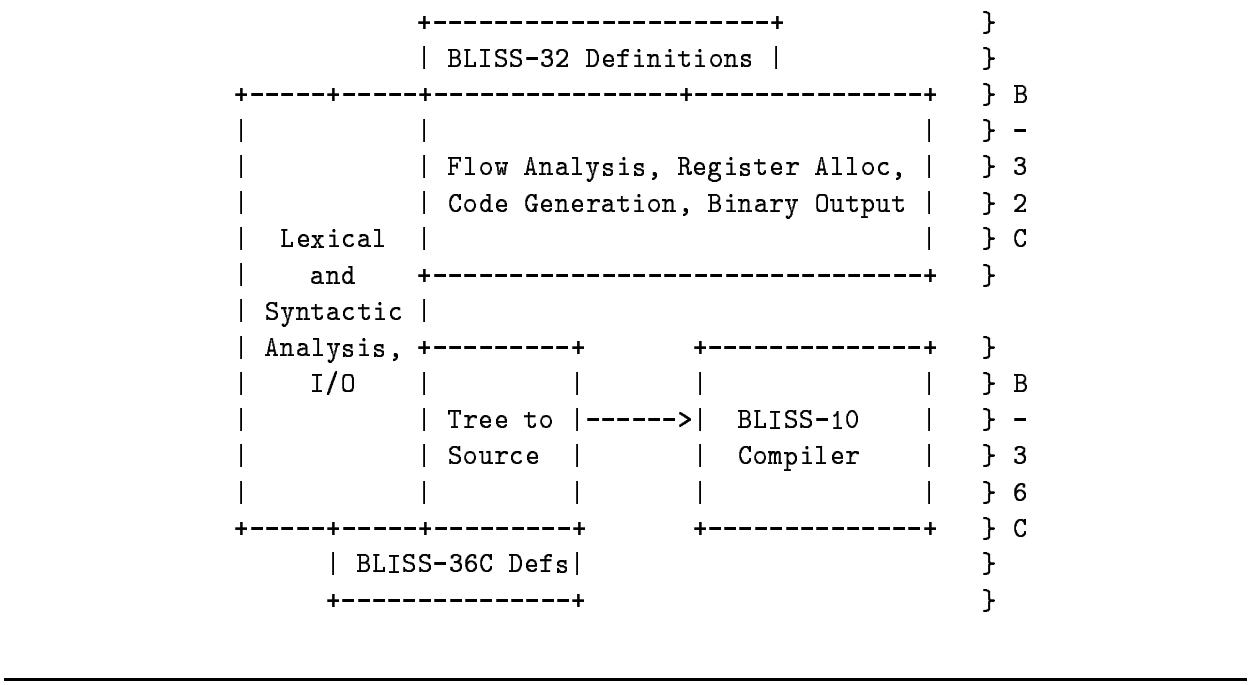


Figure 37: Structure of BLISS-36C and BLISS-32C Cross-Compilers

3.2.3 BLISS-16C

The strategy that produced BLISS-36C was effective enough that the process was next repeated to produce a comparable compiler for BLISS-16, known as BLISS-16C. In this case, the front-end, already shared between BLISS-32 and BLISS-36C, was adapted once again for the BLISS-16C front-end. The back-end of the translator was adapted to produce BLISS-11 output instead of BLISS-10.

Figure 38 illustrates how the code sharing model was extended to include BLISS-16C. Note that the back translation part now has both shared and language specific components.

The accomplishments as of June 1977 were the coordinated internal delivery of three production quality compilers supporting the new BLISS language for the PDP-10, VAX-11, and PDP-11. The main techniques that made this possible in the 21 months from conception to delivery were the extensive sharing of code among the front-ends of the three compilers and the two translators, the extensive re-use of existing code in the original creation of BLISS-32 from BLISS-11, and the hidden use of BLISS-10 and BLISS-11 as part of BLISS-36C and BLISS-16C.

3.2.4 Bootstrapping and Native Compilers

In the early stages of the BLISS-32 development, the groundwork was laid for the eventual bootstrap of BLISS-32 onto VAX-11. With BLISS compilers for the three architectures in place and with the VAX/VMS operating system for the VAX-11 sufficiently complete and stable, we were ready to put the BLISS transportability concepts to their first major test.

3.2.5 BLISS-32

The bootstrap of the BLISS-32 compiler from the PDP-10 to the VAX-11 took place in two stages. First, the compiler was transliterated from BLISS-10 to BLISS-36C. Second, the compiler was used to compile itself and was checked out on the VAX-11.

The conversion from BLISS-10 to BLISS-36C used a combination of ad hoc semiautomatic and manual methods. The bulk of the conversion was performed using a SNOBOL program written especially for this purpose. This program used a relatively simple and shallow parse of the BLISS-10 input and was able to perform all lexical conversions and most syntactic conversions, leaving about 10 to 15 percent of the task to be completed by hand.

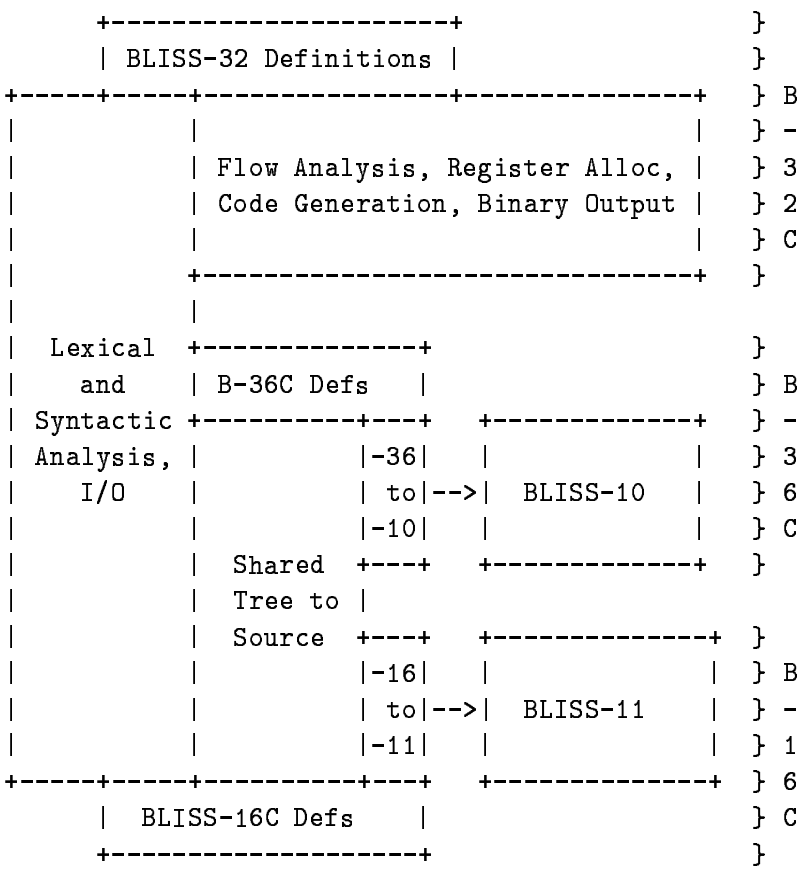


Figure 38: Structure of BLISS-36C, BLISS-32C and BLISS-16C Cross-Compilers

The converted BLISS-32 sources, now written in BLISS-36C, were next compiled by the original BLISS-32 cross-compiler using the data structure definitions for the VAX-11.

The BLISS-32 native compiler (now completely self-hosting) on the VAX-11 was internally released in December 1977, just six months after the start of the bootstrap effort. The BLISS-32 bootstrap compiler (written in BLISS-36C) was also released at the same time to replace the version written in BLISS-10.

3.2.6 BLISS-36

At the same time as the BLISS-32 bootstrap began, a project began to completely replace BLISS-36C. This new compiler, known as BLISS-36, was (of course) written in BLISS-36 using the BLISS-36C implementation as its interim basis. Moreover, the compiler architecture originally developed for BLISS-11, and refined, simplified, and extended for BLISS-32, was used by BLISS-36 to obtain better code quality than had been provided by BLISS-10.

The code sharing established in the BLISS-10 versions of BLISS-32 and the translators was preserved during the BLISS-32 bootstrap. Thus the BLISS-32 conversion provided the front-end needed for BLISS-36 as well as for itself.

The degree of code sharing was increased to include parts of the compilers that had not been relevant before. For example, the global flow analysis phase of the BLISS-32 compiler was not used at all by the translators, but was easily adapted for use by BLISS-36. Similarly, the register allocation phase, while obviously somewhat target machine specific, was also made partially shareable by both BLISS-32 and BLISS-36. After the new BLISS-32 compilers were stable and released, the new front-end, written in Common BLISS, was easily incorporated into the BLISS-36 work.

The first version of the BLISS-36 compiler was internally released in June 1978, one year after development was started and six months after the first BLISS-32 native compiler release.

3.2.7 BLISS-16

The next logical step in our development would have been to bootstrap a BLISS-16 compiler onto the PDP-11. Unfortunately, the BLISS compilers are much too large to fit into the address space of the PDP-11[23].

Instead we settled on providing BLISS-16 cross-compilers on both the PDP-10 and VAX-11. The VAX-11 was particularly attractive as a host for PDP-11 software development at that point because it included an emulation mode for non-privileged PDP-11 code. This meant that a considerable amount of PDP-11 software could be developed completely on the VAX-11.

Later, around 1981, a serious attempt was made to shoe-horn a variant of BLISS-16 [19] onto the PDP-11 using a combination of language subsetting, simplified code generation, and heavy overlaying. While the resulting compiler generally worked, it was quite limited in the size of module it could compile. The compromises were deemed too severe and the experiment was abandoned.

3.3 Methodology

In addition to its role as a BLISS language and compiler development group, the Development Methods Department was chartered to provide documentation, training and tools intended to improve DEC's software development methods more generally.

One popular idea of the time was the notion of a program formatter that could re-format a program in accordance with predetermined style guidelines. So we developed a BLISS reformatter we called `PRETTY`. It actually did quite a good job, and some people really liked it. But it never quite matched good programmer created formatting. Eventually it just seemed to be forgotten.

More enduring was the *VAX-11 Software Engineering Manual* [13]. This document began life as a codification of the BLISS coding and documentation style (which `PRETTY` was intended to help mechanize).

The module and routine header documentation style developed then has evolved little over time and is still in regular use even today for BLISS code. Similar and compatible style templates were also developed for assembler language and later C.

[23] The PDP-11 had a 64K byte virtual address space. For comparison, the BLISS-32 compiler on VAX-11 required at least a 256K byte address space and the BLISS-36 compiler on the PDP-10 required at least an 80K (36-bit) address space in typical large scale use.

This manual grew into a comprehensive project handbook that set forth the "DEC way" of doing software. It included planning, design and development documentation requirements, various review and approval processes, codification of how engineering, training, manufacturing and field support (should) work together, and so on.

This document continued to evolve long after the Development Methods Department ceased to exist, became a Corporate Standard [20], and was even used as a customer sales tool.

3.4 To Type Data or not to Type Data

Of all the things that we set out to do to and with BLISS in 1976, the one thing that never happened was to introduce data typing.

We initially thought that data typing was a prerequisite for solving many portability problems. But along the way, as we gained experience porting the BLISS compilers themselves, we learned alternatives.

We also learned some hard lessons in how difficult it can be to add features to a programming language. However much certain characteristics of BLISS-10 seemed undesirable, a simple fact was that it had a coherence and internal consistency that sometimes made apparently simple changes more difficult or far reaching than expected. One example is seen in the evolution of the `<p, s>` field access notation from a "normal operator" that produced one value from another in BLISS-10 to optional operands to the fetch and assignment operators in BLISS-11. It sounds simple and we covered it earlier in just a few words–but it fact it had subtle and far reaching effects on the meaning of structure declarations, how they were compiled and how they were used, that were still being worked out as we implemented the second generation of BLISSes.

Still, we thought that data typing could be introduced in BLISS. Both Brender and Wulf (separately) claimed that they thought it could be done "in a reasonable and consistent manner" [45]–but no proposals ever got written down. In the end, there was just too much else to do (all of which did get done) and we learned how to do without it.

3.5 The New Generation is Established

By the end of 1979, the new generation of highly language compatible BLISS compilers was firmly established at DEC. We had

- BLISS-36 hosted on the PDP-10
- BLISS-32 hosted on both the PDP-10 and VAX-11
- BLISS-16 hosted on both the PDP-10 and VAX-11

4 The Golden Years at DEC: 1978-1990

With a common BLISS language, with BLISS compilers available for all three of DEC's major architectures, and with the backing of Corporate policy, BLISS became established as the language of choice for new software development.

For the PDP-10 family, most software that had been written in BLISS-10 was eventually converted to use BLISS-36 (but not APL nor FORTRAN-10) A number of new products were initiated which were intended for both PDP-10 and VAX systems. But there was a very large base of software written in PDP-10 macro (which some strong adherents claimed *was* a higher level language) so adoption in the 36-bit world proceeded slowly. Moreover, development of the 36-bit line was canceled in mid-1983, making the matter moot.[24] Products that were written in BLISS-36 are listed in Figure 39.

Similarly for the PDP-11 family, software that had been written in BLISS-11 was eventually converted to use BLISS-16. But the lack of a native BLISS-16 compiler proved a strong disincentive to expanded usage. Further, the PDP-11 line was rapidly overshadowed by the VAX family. The PDP-11 family continued as a viable product line well into the 1990s, although software work was mostly incremental enhancement or maintenance in nature. Products that were written in BLISS-16 are listed in Figure 40.

BLISS-32 and the VAX family were created and grew up together, so it is natural that BLISS usage flourished on VAX/VMS. Figures 41, 42, 43 and 44 list many of the more widely used products and components that were written in BLISS-32.

4.1 "BLISS Extensions"

No matter how complete we thought we had been in defining BLISS, nevertheless there rapidly grew up a variety of more or less common macro and literal declarations that were used by most groups that used BLISS. A sampling of such declarations is shown in Figures 45 and 46. As can be seen, many of these declarations were simple and straightforward, reflecting minor notational convenience and/or esthetics.

[24] The development of a common implementation language between the PDP-10, PDP-11 and VAX-11 provided a vehicle for building a variety of personal and professional ties amongst the compiler development teams in the several groups. Following cancellation of the 36-bit line, proportionately many more of the compiler folks from that group were attracted to and joined the remaining groups compared to the hardware and operating system teams.

APL-10/20	APL compiler/interpreter
BASIC-PLUS-2	Basic compiler/interpreter
BLISS-16	BLISS-16 cross-compiler
BLISS-32	BLISS-32 cross-compiler
BLISS-36	BLISS-36 compiler
COBOL-20	COBOL Compiler
DBMS-10/20	Database Management System
DAP	DECnet Data Access Protocols
DIL	DECnet Interchange Library
EDT	Editor (ported from VMS)
MX	DECnet Mail eXchange
RMS-20	Record I/O Management System

Figure 39: PDP-10 Products Written in BLISS-36

DSR	DEC Standard Runoff (text formatter)
FORTRAN IV-PLUS	Optimizing FORTRAN compiler
MCB	DECnet Multi-Communications Base support

Figure 40: PDP-11 Products Written in BLISS-16

CDU	Command Definition Utility
DEBUG	Multilanguage debugger
DTK	DECtalk shared library
EDT	Editor
F11ACP	Files-11 file system control program
ISAM	Indexed Sequential I/O Subsystem
LIBR	Object and text library manager
LIB$	General purpose shared run-time library
LINK	Object File Linker
MACRO	Macro assembler
MAIL	Mail utility (pre V7)
MSG	Message Utility
PCA	Performance and Coverage Analyzer
SMG$	Screen Management shared library
STR$	String management shared library

Figure 41: VAX/VMS Tools and Utilities Written in BLISS-32

CDD	Common Data Dictionary
CMS	Code Management System (version control)
DSR	Digital Standard RUNOFF
DTM	Test Manager (regression testing)
MMS	Module Management System (make-like builder)
NOTES	Newsgroup-like utility
SCAN	Macro processor (compiled)
SCA	Source Code Analyzer (source browser)
SPM	Software Project Manager
TPU	Text Processing Utility (basis for LSE and other editors)

Figure 42: Software Development Tools and Utilities Written in BLISS-32

`Ada`	Ada 83 Compiler and library
`APL`	APL compiler/interpreter
`BASIC`	BASIC compiler/interpreter and library
`BLISS`	Multiple BLISS compilers
`COBOL`	COBOL compiler and library
`FORTRAN`	FORTRAN 77 compiler and library
`Pascal`	Pascal compiler (V2 and later) and library

Figure 43: Compiler Products Written in BLISS-32

RDB	Relational Data Base system
ALL-IN-1	Integrated Office Systems (not all but major parts)
DFS	Distributed File System
MAILBUS	X.400 based mail transport system (V1)
OPS/5	Artificial Intelligence system

Figure 44: Other Digital Products Written in BLISS-32

```
!               "Keyword" definitions
!
literal
    null        = 0;

! Macro to do nothing -- a place holder
!
macro
    skip        = 0 %;

! Define ALIAS keyword for BLISS-32V, which doesn't have it
!
%if %bliss(BLISS32V)
%then
    macro
        alias   = volatile %;
%fi

!               Control structures
!
macro
    loop        = while 1 do %,
    elsif       = else if %;
```

Figure 45: Typical "BLISS Extensions"–Part 1

```
! and_then(B1,B2,...Bn) returns true if all of Bi are true.
! Evaluates the Bi left to right and stops on the first false one.
!
macro
    and_then(B)[]=
        %if %length eql 1
        %then
            (B)
        %else
            (if (B) then and_then(%remaining) else false)
        %fi %,

! or_else(B1,B2,...Bn) returns false if all of Bi are false.
! Evaluates the Bi left to right and stops on the first true one.
!
    or_else(B)[]=
        %if %length eql 1
        %then
            (B)
        %else
            (if (B) then true else or_else(%remaining))
        %fi %;
```

Figure 46: Typical "BLISS Extensions"–Part 2

Most projects started with a BLISS extensions file from an earlier project (the file name `BLISSEXT.REQ` was popular) and evolved it to fit that project's needs. Unfortunately no effort was ever made to "standardize" even a core subset of BLISS extensions across groups, and in later years the resulting incompatibilities sometimes became an impediment to sharing and reusing code among groups.

Some projects elaborated the BLISS extensions in far more exotic ways. One example of this is discussed in the following Section.

4.2 Compile-Time Processing and VAX Ada

The VAX Ada project began at Digital in 1979 to monitor developments in the DoD Ada Language definition activity and to explore implementation issues. In 1981, product development began in earnest and in 1984 Digital delivered the first production quality Ada implementation. It was written in BLISS-32. One aspect of the VAX Ada implementation helps illustrate how pervasive and fundamental the BLISS compile-time mechanisms were in the use of the language.

The internal representation of Ada was an abstract syntax tree, which described the entire compilation unit, including declarations as well as statements. This representation was created by the parser, then refined and annotated by multiple phases that performed static semantics and dynamic semantics analysis and finally discarded after code generation.

The abstract syntax tree representation was defined using a set of BLISS macros, which ended up serving many purposes. To get the flavor of the definitional style, consider Figure 47. There is a lot of information packed into this definition, which we summarize very briefly:

- The first line begins the definition of a node **KI**nd named `block_stmt`,
  - whose BLISS structure name is therefore `N_BLOCK_STMT_STR`,
  - which has a source program locator field named `N_BLOCK_STMT_LOCATOR`, and
  - has an embedded linked list pointer named `N_BLOCK_STMT_NEXT`,
  - and this kind of node is *denotable*, that is, can be pointed to by nodes other than just its parent.

- The next several lines declare fields that correspond to the Ada **GR**ammar, and are therefore filled in by the parser. The comments to the right document the corresponding part of the grammar involved. Grammar fields, as well as the subsequent non-grammar ATtributes, have several properties:
  - A field name, such as `N_BLOCK_STMT_STMT_HEAD`, `N_BLOCK_STMT_PRAG_REP_CC`, and so on;
  - The *type* of field, such as `HEAD` indicating the start of a linked list using the `NEXT` fields of other nodes, `FLAGS_L` indicating a **L**ongword (up to 32 bits) of flags, `SYMTAB` indicating a pointer to a code generator symbol table entry, and so on;
  - Various other properties, such as `following` indicating that the flag definitions follow (see the FLAGS macro after the end), or `SPP` indicating that the field is a *software paging point*, that is, a pointer to another node that may either be in main memory or may still be represented on disk.
- The last (non-indented) line indicates the **EN**d of the node description.
- The `FLAGS` macro defines the actual flags that occupy the `BLOCK_FLAGS` field; they are named `NV_BLOCK_FLAGS_HAS_OPTIMIZE` and so on.

In short, there is a rich, compact and very target application specific data definition language that is used to define the compiler internal representation.

```
KI( block_stmt, LOCATOR, NEXT, DENOTABLE    )! block statement
GR(   DECLS                                 )! declare [ DECLS ]
GR(   STMT_HEAD,        HEAD                )!   begin STMT ;...
GR(   HANDLER_HEAD,     HEAD                )!   [ exception
                                             !       HANDLER ;... ]
GR(   END_TOKEN                             )!   end [ TOKEN ] ;
AT(   BLOCK_FLAGS,      FLAGS_L,  following )!
AT(   PRAG_REP_CC,      CC_ATTR,  , SPP     )!
AT(   BEGIN_LOCATOR,    LOCATOR             )!
AT(   SIGARGS,          SYMTAB              )!
AT(   SAVED_OPT_STATE,  ENUM_B,   NFVD_OPT  )!
AT(   SPARE1,           HEX_W,    , SPARE   )!
EN()

        FLAGS(  BLOCK_FLAGS,
                        ! For pragmas
                        HAS_OPTIMIZE,
                        !
                        DST_OPEN,
                        DST_HAS_SEG)
```

Figure 47: VAX Ada Tree Definition Sample

The TREEDEF.REQ source file, which consists mainly of definitions like those shown, is included in three different compilations and is actually processed five times, by five different sets of macro definitions. In each case, a set of macros is declared having the names KI, GR, AT, and so on; where this source file is processed twice in the same compilation, the initial macro definitions are undeclared[25] and given another declaration for use during the second pass.

- Compilation of TREE.REQ resulted in a compiled library file that defined the structures, field-sets, macros, and sets of enumeration codes that are included in the compilation of the rest of the VAX Ada compiler sources.
- Compilation of TREE.BLI resulted in creation of a *node property table* which provided the compiler with a complete run-time description of its own internal representation. This table was used heavily by the compilation library manager subsystem, for example, to read and write compilation units to disk storage. It was also used by tree copying algorithms that were central to the implementation of generic instantiation, procedure call inlining, and default parameters. This allowed changes in the tree definition to be made in one central place and be automatically incorporated in these other uses without even recompilation, let alone source modification. A number of self-checking internal consistency checking modules also depended heavily on the node property table.
- Compilation of TREEDEB.BLI extended the node property table to include the text of field names and decoding tables for converting flags and enumeration codes into source names. This information was used by tree interpretation and display that were part of the integrated debugging tools within the compiler.

The node property table provided the basis for a variety of debugging display routines that were written as part of the project. These routines go well beyond what could be achieved by even the best general-purpose debugger, including VMS DEBUG. Rather than show the tree as a series of independent nodes, we displayed the tree as the kind of nested data structure it really is. Extraneous information, such as the addresses of nodes that are not legitimately pointed to by attribute pointers, are suppressed. Certain kinds of nodes that are actually "private" in the Ada sense can be displayed in a natural manner by display routines created as part of the implementation of these abstract node kinds.

[25] Recall that `undeclare` is a BLISS "declaration" that erases a prior declaration of a name, so that the name becomes available to be declared again.

Figure 48 illustrates one such display for a simple example program. It is not necessary to understand this display in any kind of detail. Rather, you should mentally compare the kind of display you could get from node-by-node displays versus the highly annotated and interpretive example shown. A display tool such as the one that produced this example clearly is application specific and could only be produced as part of the project in which it is used. But, more importantly, even a project-specific tool such as this would not be practical without the run-time self-description of the data structures in use.

The BLISS compile-time facilities proved extremely effective at automating key parts of the VAX Ada compiler implementation. Certainly a separate project written tool could have been written to accomplish the same thing, but it simply was not necessary.

For additional description of how BLISS was employed in the VAX Ada compiler, see Brender [7].

4.3 GEM

Most of the compilers created by DEC during the Golden Years incorporated their own unique optimization and code generation capabilities. The exception was code generation technology originally brought inhouse for VAX C and VAX PL/I and then later also used for VAX Ada. The benefits and limitations of this first attempt at a common shared code generation technology led to the initiation of the GEM project, whose goal was the creation of a rapidly retargetable highly optimizing compiler backend [3]. GEM was originally targeted to a new DEC RISC processor intended to supersede the VAX. However, that processor never came to market.

GEM was originally written in BLISS-32. As a result, BLISS became a critical tool supporting the emergence of the highly performance oriented GEM-based compilers. For targets other than VAX, the BLISS front-end was reworked to use GEM optimization and code generation. The relationship is now so close and the two are so interdependent that both GEM and BLISS are managed together as a single development project.

```
    1   procedure FOO is
    2       X : INTEGER := 0;
    3   begin
    4       if X > 0 then X := X + 1; end if;
    5   end;

0088D3B0:
     FOO ˜ K_PROC_BODY_DECL<ANA_USED @1:0
        <<void>>:
        <<void>>:        _CORRES_BLOCK
        K_BODY<
008AD9F4:   K_DECLS< @2:4
008AD898:'      X ˜ K_OBJ_VARIABLE<NON_CONST, ANA_USED @2:4
          .       INTEGER ˜ K_REFER<STATIC 0000D230: @STANDARD/4>
          .       0 ˜ K_INTEGER_VAL<CTC_VAL, STATIC @2:22
          .          SL>
          .       CONF_BEG_SEQ     _OBJ_FLAG1
          .       OCCURS_IN_NAME   _OBJ_FLAG2
          .       008ADDE0:        _SYMTAB
          .       <null list>      _PRAG_REP_CC>
          .    <no flags set>      _DECLS_FLAGS
          .    <<void>>:           _CONTINUE
          .    00000000   _CL_VIS>
           <
          .    K_IF_STMT< @4:4
          .       K_BINARY_OP<NON_CONST @4:9
          .          GT_INT
          .          X ˜ K_REFER<NON_CONST 008AD898: @2:4>
          .          0 ˜ K_INTEGER_VAL<CTC_VAL, STATIC @4:11
          .             SL>
          .          0000D230: INTEGER @STANDARD/4   _BINDING
          .          0000D140: BOOLEAN @STANDARD/2   _RES_TYP>
          .       <
          .          K_ASSIGN_STMT< @4:20
          .             X ˜ K_REFER<NON_CONST 008AD898: @2:4>
          .             K_BINARY_OP<NON_CONST @4:25
          .           .    BINARY_PLUS_INT
          .           .    X ˜ K_REFER<NON_CONST 008AD898: @2:4>
          .           .    1 ˜ K_INTEGER_VAL<CTC_VAL, STATIC @4:27
          .           .       SL>
          .           .    0000D230: INTEGER @STANDARD/4   _BINDING
          .           .    0000D230: INTEGER @STANDARD/4   _RES_TYP>
          .             <no flags set>    _ASSIGN_FLAGS>
          .       <void list>
          .       <null locator>          _LAST_LOCATOR
          .       <no flags set>          _IF_FLAGS>>
           <void list>
           <<void>>:
           NO_EXCP_PART  _BODY_FLAGS
           <null list>   _PRAG_REP_CC
           FF7529B8:     _ZONE
           @3:0          _BEGIN_LOCATOR
           @5:0          _LAST_LOCATOR
           0088CA1C:     _SIGARGS
           TIME          _SAVED_OPT_STATE
           TIME          _LOCAL_OPT_STATE
           <null list>   _LOCAL_SUPP_CC>
        DST_HAS_SEG, DST_HAS_ZEM, EXIM_ALLOWED, IS_ELABORATED, IS_GBL_VIS,
           IS_LIB_UNIT, MECH_FIXED      _PROC_FLAGS
        <null list>      _PRAG_REP_CC
        0088CA1C:        _SYMTAB
        <<void>>:        _FULFILLS
        <<void>>:        _STATUS_OBJ>>
```

Figure 48: Example "Automated" Tree Display in the VAX Ada Compiler

Property	MIPS
Word size	32
Addressable unit	8-bit byte
Byte pointer	(just an) address
Byte access	signed or unsigned byte and word loads
Registers	32
	(special purpose: 0, 1, 26, 27, 28, 29, 31)
Floating point	16 separate floating point
	(double precision in even/odd pair)

Figure 49: Summary of MIPS Properties

5 The MIPS Years: 1989-1993

For a brief period from about 1989 through 1993, DEC offered a line of workstation and server products based on the MIPS R3000 architecture [26]. (The summary of properties in Figure 49 can be compared with those in Figures 12 and 18.) While this was the first RISC architecture supported by BLISS (unless you count the PDP-10), in most respects the machine architecture had little impact on the language. In fact, many of the changes were influenced by the Ultrix[26] operating system environment more than hardware considerations.

If anything, this period might well be regarded as an interlude that provided "practice" for the DEC Alpha architecture that followed. A summary of the few language changes that did result is shown in Figure 50.

BLISS-32M, as it was called, was never used much even within DEC. It was ported to run on the DECstations, but was never released outside DEC. Its primary use was to port the DEC FORTRAN and DEC Ada products from VAX/VMS to Ultrix. (All three compilers used the GEM backend.)

[26] DEC's version of UNIX

- The EXTERNAL_NAME attribute was defined for use on data or routine declarations that are either imported (EXTERNAL) or exported (GLOBAL); it specifies the case sensitive name to be used for the entity in the object language for the module.
- All of the BLISS-32 linkage types were allowed but treated the same; a new UNCOUNTED linkage was added which was like the others except that no argument count was passed on a call (the argument count was passed in register 15, normally a temporary, non-preserved register).
- The BLISS-32 linkage functions, ACTUALPARAMETER, ACTUALCOUNT, NULLPARAMETER and ARGPTR are supported.
- The only builtins included from BLISS-32 were those for rotate and shifting: ROT, SLL, SRL and SRA.
- For floating point:
  - The floating point registers were not accessible.
  - Floating point values could only be passed by reference.
  - New %S and %T lexical forms replaced %E and %D to specify IEEE floating point literals.
  - Builtin functions corresponding to the MIPS floating point arithmetic and comparison operations replaced those for VAX: ADDS, ADDT, SUBS, SUBT, MULS, MULT, DIVS, DIVT, CMPS, CMPT, CVTLS, CVTLT, CVTSL, CVTTL, CVTIS, CVTIT, CVTSI, CVTTI, CVTST, CVTTS, CVTTS, CVTRSL, CVTRTL.
- The data attributes ALIGN, VOLATILE and SHARED were allowed between the ref and a structure name in a structure attribute to indicate that they applied to the pointee data segment rather than the pointer.
- Run-time support libraries were used to support
  - The character handling functions
  - The exception handling mechanism (SIGNAL and ENABLE, which had to be layered on the UNIX signal mechanism)
- The %BLISS lexical function accepted BLISS32V and BLISS32M. %BLISS(%BLISS32) was true in both dialects. %BLISS(%BLISS32V) and %BLISS(%BLISS32M) were true only in BLISS-32 VAX and BLISS-32 MIPS, respectively.
- A SYNTAX_LEVEL switch (both source and command line) was added to allowed introduction of new keywords
- The keywords QUAD and SHARED were reserved for anticipated 64-bit processor and multi-threading support; these were never implemented.

Figure 50: BLISS MIPS Differences from BLISS-32 VAX

Property	Alpha
Word size	64
Addressable unit	8-bit byte
Byte pointer	(just an) address
Byte access	initially long and quad only later byte and word added
Registers	32 (special purpose: SP, R31)
Floating point	32 for floating point (both VAX and IEEE supported)

Figure 51: Summary of Alpha Properties

6 The Alpha Years: 1990-

The Alpha 64-bit architecture, in the form of the 21064 chip and systems that used it, was introduced in late 1991. It is a RISC, load/store, base-register oriented addressing architecture in contrast with the CISC, multiple operands with complex addressing mode characteristics of the VAX architecture. (The summary of properties in Figure 51 can be compared with those in Figures 18 and 12.) Despite these differences, the impact on the BLISS language was quite small and straightforward–in most respects.

6.1 Language Evolution

The most interesting aspect of BLISS for Alpha is that there is not just a new 64-bit variant of the language, BLISS-64, as might be expected. There was a new 32-bit variant as well. More interesting yet is the mixed 32-bit/64-bit hybrid that eventually emerged.

First, we'll cover the most direct hardware influences and then consider what are more properly viewed as operating system environment influences.

6.1.1 BYTE, WORD, LONG and QUAD

For compatibility with the VAX, Alpha retained the terms *word*, *longword* and *quadword* to mean two, four and eight bytes, respectively.[27] The quadword sized value became the size of all values in BLISS-64, of course. The keyword QUAD was adopted to indicate a value of this size.

6.1.2 Floating Point

Alpha floating point includes

- 32 floating point registers (separate from the 32 general purpose registers)
- Direct support for multiple data formats

S-float	32-bit	IEEE compatible single precision
T-float	64-bit	IEEE compatible double precision
F-float	32-bit	VAX compatible single precision
G-float	64-bit	VAX compatible double precision

(There was also more limited support for D-float, a 64-bit PDP-11 and VAX compatible single precision representation. The VAX 128-bit H-float has no hardware support at all, but a 128-bit IEEE compatible X-float is supported in software by several programming languages.)

As with BLISS-16 (and BLISS-11 before it), BLISS-64 provides no direct access to the floating point register set. However, there is a full complement of machine specific builtin functions that allow use of the hardware functionality. There are also the expected multiple lexical constant notions for creating literal values of each of the several data formats (for example, `%S'1.0'`, `%T'-0.1'` in addition to the earlier `%F`, `%G` and `%D` on VAX).

In addition, there are some unusual builtin operations that made it possible to make calls that passed and returned values using the floating point registers. Some examples are given in Figure 52. However, there is no way to write such routines in BLISS.

[27] Quadword sized values had very specialized uses in the VAX architecture and did not get reflected in BLISS.

```
! Call printf, passing a control string, an integer and a
! T-float parameter
!
external routine
    printf : novalue;
macro
    tfloat = vector[8, byte] %; ! Pseudo type name
local
    float  : tfloat;

printf(string, .integer, %tfloat(float));

! Call math$ccos, which takes an F-float complex parameter
! passed by value and returns an F-floating complex value
!
external routine math$ccos;
macro
    fcomplex = vector[2, LONG] %,
    real(x)  = x[0] %,
    imag(x)  = x[1] %;
local
    input    : fcomplex,
    output   : fcomplex;

math$ccos(%ffloat(real(input)),  %ffloat(imag(input)),
          %ffloat(real(output)), %ffloat(imag(output));
```

Figure 52: BLISS-64 Floating Point Special Function Examples

6.1.3 Alignment and Granularity

The Alpha architecture requires natural alignment of the memory address in load and store operations. For example, the address of a load long (`LDL`) instruction must be a multiple of 4, the size of a longword. Sequences using the special load unaligned and store unaligned (`LDQ_U` and `STQ_U`) instructions made accesses possible without taking a hardware trap, but are far too expensive to use routinely.

Memory operands on the VAX did not need to be aligned, although the hardware was more efficient if they were. This meant that some software being ported from VAX to Alpha might suffer performance degradation due to poor alignment.

To facilitate better interfacing with this aspect of the hardware, BLISS-64 allowed the `ALIGN` attribute to be used on a few more declarations, most importantly, the `EXTERNAL` declaration to provide information to the compiler as to what alignment could be assumed.

More challenging from a compiler implementation perspective was the fact that the Alpha architecture omitted instructions for word and byte access. The *granularity*, that is, the smallest unit of memory access, was the longword. This was later corrected beginning in the 21164A (also known as EV56) processors. Granularity issues did not influence the BLISS language design as such and is not further considered here.

6.1.4 BLISS-32 *and* BLISS-64

When the Alpha architecture was first being designed, it was expected to support two operating system environments: a 64-bit version of UNIX and a 32-bit version of OpenVMS. Because OpenVMS/VAX was such a huge part of DEC's installed base, it was felt that everything possible must be done to make porting from OpenVMS/VAX to OpenVMS/Alpha as straightforward as possible while retaining the ability to extend OpenVMS with full 64-bit support later. Thus, a 32-bit variant of BLISS (BLISS-32E as it came to be called) was needed for Alpha as well as a 64-bit variant (BLISS-64E).

BLISS-32E also came to be used on UNIX in conjunction with the *taso* option, in which a program can be compiled and linked for execution in only the lowest 32-bit segment of the Alpha address space. This was strictly a software convention–no special hardware or operating system modes were involved.

The UNIX taso option for Alpha is valuable in two cases:

- For pointer intensive applications (compilers come to mind) that do not require more than a 32-bit address space. Using 32-bit pointers reduces the application size, which helps performance.
- For C/C++ programs that have inappropriate assumptions about the relative and absolute sizes of pointers, `int`s and `long`s too strongly embedded to allow porting to a system with 64-bit pointers and `long`s and 32-bit `int`s.

The taso support did not attempt to create a "pure" 32-bit environment, but it was complete enough to be quite valuable nonetheless.

For a number of years, most of DEC's compilers on UNIX themselves operated in the taso environment, were built using BLISS-32E, but produced code that operated in the full 64-bit environment.

6.1.5 The BLISS-32/64 Hybrid

The problem with producing a 64-bit application using a compiler operating in the 32-bit taso domain is that BLISS-32E is truly 32-bit, which is obvious on the one hand and has fundamental implications on the other.

- At the simplest level, it means that internal compiler data structures are limited to sizes representable in 32-bits. This limits even the *description* of entities that made use of more than 32-bits of storage, such as FORTRAN COMMON blocks or C global variables.
- To perform compiletime constant arithmetic, even input conversion of literals, required the same sort of awkward techniques that are required to cross-compile from a 32-bit to a 64-bit environment.
- Finally, even though the registers were 64-bit wide, algorithms that should benefit from that width (small sets, bitmask techniques for memory management, searching, or the like) do not benefit from the available width in the real environment.

The UNIX compilers were originally developed and cross-compiled on 32-bit VAX/VMS, which meant that these limitations were viewed as a necessary concession. But, full 64-bit clean support

was wanted for the subsequent natively hosted compilers. Was there a way to have 32-bit pointers using BLISS-64?

A new command option `/ASSUME=LONG_DEFAULT` and the module switch `LONG_DEFAULT` were introduced, which have the following effects:

- `LONG` is the default alignment and size and `SIGNED` the default sign for scalars. This applies to scalars that have neither allocation attribute nor sign attribute, and to those that are declared `LONG` but have no sign attribute.
- `VECTOR` defaults to `VECTOR[,LONG,SIGNED]`. This applies to both `VECTOR[`*n*`]` and `VECTOR[`*n*`,LONG]`.
- `BLOCK` defaults to `BLOCK[,LONG]`.
- `BLOCKVECTOR` defaults to `BLOCKVECTOR[,,LONG]`.
- So-called *anonymous fetches*, for example `.(A)`, are longword fetches.
- So-called *anonymous stores*, for example `.A = 1`, are longword stores.
- Data segments that are declared as `ref` structures are `SIGNED LONG`.
- However, `%UPVAL` remains 8 and `%BPVAL` remains 64.

The effect is to create a compilation mode in which addresses are stored and used as 32-bit values but otherwise the full 64-bit wordsize is usable[28].

Through use of these switches in combination with some adjustments in the GEM macros that declare (pseudo-)types, it was possible to compile the same sources on

- VAX/VMS using BLISS-32V to create a BLISS-32E cross-compiler (VAX/VMS to Alpha Tru64 UNIX) with 32-bit limitations
- VAX/VMS using the BLISS-32E cross-compiler to create a BLISS-64E native compiler with 32-bit limitations
- Alpha Tru64 UNIX using BLISS-64E to create a clean native-compiler (without 32-bit limitations)

[28] There are some additional effects that are outside the scope of this paper.

```
! If success, assign target address and return true
! Else return false
!
routine TRYIT (TARG : ref TARGET_ADDRESS) =
    begin
    ...
    TARG[] = ...;
    ...
    end;

routine WORKER =
    begin

    local
        TEMP;

    if TRYIT(TEMP) then ...;

    end;
```

Figure 53: BLISS-32/-64 Hybrid Usage Illustration

6.1.6 Tighter Checking

While the `LONG_DEFAULT` and `REF_LONG` conventions accomplished what we intended, they also proved error prone in a particularly awkward way. To understand the problem, consider the schema shown in Figure 53.

There are three cases to consider:

Compiler	Host BPVAL	Target BPVAL	Bits assigned to TEMP
BLISS-32V	32	32	32 of 32
BLISS-64 Hybrid	64	32	64 of 32 (!)
BLISS-64	64	64	64 of 64

In the first and third cases, everything will work as intended. However, in the second case, routine `TRYIT` may compute a value larger than 32-bits which is assigned to `TEMP` which is only 32-bits (according to the `ASSUME_LONG` convention). The result is to store beyond the allocation of TEMP, which may or may not have visible consequences.

To help with this problem, these changes were introduced into BLISS (in 1999):

- Parameter lists were allowed on external routine and forward routine declarations (so that all declarations of a routine name could share the same parameter information).
- The compiler option `/CHECK=SHORT_ADDRESS` results in diagnostics for
  - fetches from address that are less than 64 bits,
  - stores into addresses of less than 64 bits, and
  - stores of addresses into fields of less than 64 bits.
- The compiler option `/CHECK=PARAMETERS` results in diagnostics from limited checking of parameter lists of routine calls: if a formal is declared as a `ref` structure and the corresponding actual is an undotted data segment, then the segment size must not be smaller than the address size.

Note in particular that most of these checks can only be made when at least one of the entities involved is an undotted data segment name or undotted field access. Nonetheless, even this limited checking has proved quite beneficial.

6.1.7 Alpha Linux Influences

Recently (1999), BLISS was ported to the Alpha Linux environment. This was done as part of making the GEM-based FORTRAN 90, C and C++ compilers from Compaq Tru64 UNIX also available on Alpha Linux[29]. No language changes were needed as a result of this port.

[29] The most significant work from the GEM perspective was adding ELF object file and DWARF2 debugging symbol table support.

6.2 Compiler Implementation

The BLISSes for Alpha used GEM for optimization and code generation, while GEM was written in BLISS (BLISS-32 for VAX). This presented a classic compiler bootstrapping problem for which the approach was traditional and straightforward. The steps involved have already been discussed in Section 6.1.5.

Property	IA-32 (32-bit)
Word size	32
Addressable unit	8-bit byte
Byte pointer	(just an) address
Byte access	byte/word can be signed/unsigned
Registers	8 (various special restrictions)
Floating point	8 (80-bit) floating point (organized as a stack)

Figure 54: Summary of IA-32 (32-bit) Properties

7 The Intel Years: 1995-

In the mid-1990s, GEM was retargeted to support the Intel IA-32 architecture in order to offer a DIGITAL Fortran 90 product under Windows 95 and Windows NT. The original DIGITAL Visual Fortran (DVF)[30], a joint DEC and Microsoft venture that replaced Microsoft's Powerstation Fortran, was released in early 1997.

That meant porting BLISS as well, of course.

Unlike most earlier BLISSes, BLISS-32IW (as it came to be called) does not attempt to provide complete and intimate access to the full IA-32 architecture. As a practical matter it only needs to be complete enough to support the GEM backend part of the CVF product[31].

As a result, support for the IA-32 architecture has had almost no impact on the BLISS language as such. The language supported is essentially the BLISS-32E language minus the builtin functions that are specific to Alpha plus (all of) the builtins defined for MIPS. A few more specific details are summarized in Figure 55.

[30] Now known as Compaq Visual Fortran (CVF).

[31] The Fortran 90 front-end as well as the supporting run-time library is written in C.

- Floating point support matches that developed for MIPS (rather than modeling the IA-32 floating point stack hardware). Single (`%S`) and double (`%T`) precision IEEE floating point literals , as well as the arithmetic and conversion builtin functions, are supported.
- Several linkage types are provided:

`STANDARD`	Arguments pushed on stack in right-to-left order, removed by callee, counted by default
`CDECL`	Arguments pushed on stack in right-to-left order, removed by caller (normally used only for C `varargs` functions), counted by default
`FASTCALL`	Like `CDECL` except that the first two arguments (which must not be floating point) are passed in registers, uncounted by default

- The normal linkage functions are supported.
- Global registers are simulated by a 32-element vector in memory. This is the only dialect in which there are more "global" registers than architectural registers. (This was done to facilitate porting of code from architectures that are not so register poor as the IA-32.)
- Register declarations that name a specific register are supported. Register declarations that do not name a specific register are interpreted as "register preferred" (which gives them higher weight in competing for a register but does not guarantee allocation to one).
- Lexical functions `%HOST` and `%TARGET` allow the name `INTEL`.

Figure 55: Notes on BLISS-32 for the IA-32 Architecture

8 BLISS Futures

The BLISS language is continuing to expand and evolve even as this paper is being written. While it is premature to talk about specifics and/or products, the following two directions are noteworthy.

8.1 Non-Stop for Alpha

Compaq has publicly announced that the next generation of Tandem Non-Stop products will be based on the Alpha architecture, beginning specifically with the so-called EV7 Alpha chip. The Non-Stop and GEM teams are working together to create the needed compilers, which use the Tandem front-ends in combination with the GEM back-end.

The current Non-Stop product set uses big-endian data representation[32]. For backwards compatibility, the Alpha based products will use big-endian representation as well.

All BLISSes and all GEM code generators to date have supported machines that use little-endian data representations (including the MIPS-based DECstation products)[33,34]. To support the Alpha Non-Stop products, Compaq is adapting BLISS and GEM to handle the big-endian representation.

The principal question for BLISS is how to number the bits and bytes of a BLISS value to provide consistent and compatible semantics, and minimize the difficulty of adapting and porting the BLISS code for GEM. After considering possible alternatives, it was concluded that the most viable scheme is to simply reverse the order of both bits and bytes; this matches the classic big-endian conventions. The result is illustrated in Figure 56, which can be compared with Figure 1.

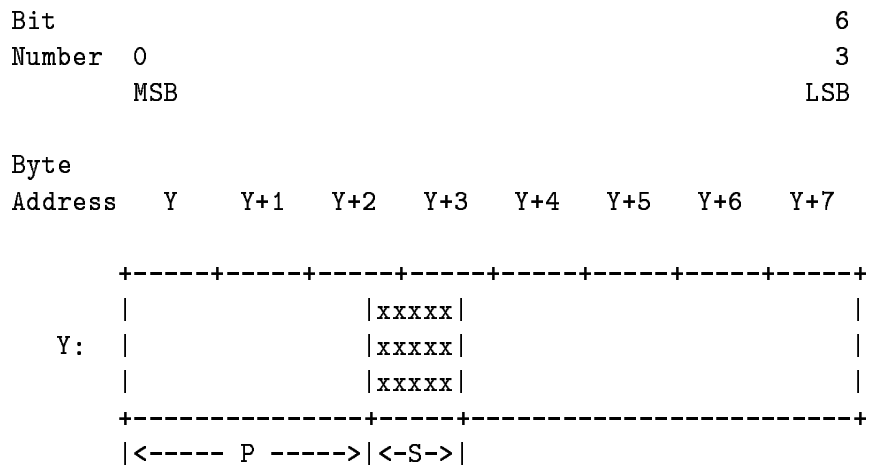
8.2 Windows NT for IA-64

As noted earlier, Compaq Visual Fortran is a very successful product for Compaq on the Windows NT operating system for IA-32. For many reasons, it would be attractive to extend that product to the 64-bit Windows NT on the Intel IA-64 architecture as well. But to do that requires BLISS-64 on IA-64.

[32] For the classic description of big-endian versus little-endian issues, see [9].

[33] Notice that while the PDP-10 bits are numbered in a big-endian style, the position and size values of a byte pointer are inherently little-endian in function.

[34] The DEC floating point representations on the PDP-11, VAX and Alpha are arguably not pure little-endian but a sort of hybrid. Given the limited role of floating point in BLISS, this is of little consequence.



where

MSB indicates the most significant bit and byte
LSB indicates the least significant bit and byte

Figure 56: Big-Endian Field Layout for Alpha

This is in progress. As is true of BLISS-32 for IA-32, BLISS-64 for IA-64 only needs to be complete enough to support the GEM backend part of an anticipated Compaq Visual Fortran product.

9 Retrospective Assessment

The beginning of this paper identifies a number of BLISS characteristics that make BLISS unusual and interesting in the family of program languages. With some thirty years of usage experience to draw on, it is appropriate to attempt an assessment of those characteristics–most importantly to suggest which have been positive and deserve consideration by future languages versus which are of little consequence or even harmful.

9.1 Typelessness

On the one hand, typelessness is a undoubtedly a limitation. On the other hand, in many ways it is usually not all that big a deal!

The effect of typelessness has been most strongly felt when trying to deal with scalar quantities that are more than the machine word size. Thus, on the PDP-11 it was necessary to adopt an awkward coding style to deal with 32-bit integers or floating point values. This was also true when dealing with 64-bit entities using the 32-bit variants of BLISS on the Alpha architecture. This property, for example, has more-or-less completely prevented BLISS from gaining any significant usage in mathematical application domains. Ever since the PDP-11, floating point has been the sore point that most severely challenged the BLISS data model.

Note that it is not typelessness as such that leads to this difficulty, but the size limitation resulting from the single size of all data.

Reflect on the way that the `BYTE` and `WORD` attributes, for example, interact with the fetch and assignment operators and you can readily view them as a kind of (very weak) data typing. The introduction of the `SIGNED` and `UNSIGNED` properties might also be viewed as a kind of weak data typing, but in fact are just the opposite: these properties really serve to wash out the size differences and normalize values to the one and only word size.

It is sometimes interesting to speculate on how differently BLISS might have evolved if *size* (and only size) had been formalized as an intrinsic part of each BLISS value. That perhaps weakest of all possible type properties, all by itself, would have eliminated many of the limitations of BLISS in dealing with "large" values, compared to the fixed size nature it has today.

9.2 Structures

The introduction of the `ref` option for structure attributes in the second BLISS generation solved a major usage awkwardness.

BLISS structures actually work quite well for aggregates that really are algorithmic in nature (arrays, matrices, even many spare variants). But they don't really work quite right for simple static arrangements–it just feels wrong to need to compose simple records (`structs` if you will) out of a combination of predefined structures and a bunch of macro and/or field definitions.[35]

Perhaps more importantly, the structure and supporting mechanisms, fundamental and flexible though they are, do not go far enough to provide full separation between the access notation and the actions to be performed. For example, there is no way to define a field composed of several disjoint parts yet access it, both for fetching and assignment using the same uniform notation. Here it would seem that at least fetch versus assignment usage context must be made available in the definitional mechanism–it is not[36].

9.3 Map and Bind Declarations

The map declaration is almost a special case of a bind declaration and thus might be considered close to redundant. It does have the advantage of not creating an aliasing problem for optimization purposes. It was used a lot in BLISS-10 and BLISS-11 where routine formal parameters did not allow structure attributes; `map` was used in the routine body to apply the appropriate structure. This was "corrected" in Common BLISS, after which the map declaration seems not to be much used.

[35] Even as we evolved and introduced Common BLISS at DEC, we reserved the identifier `record` as a keyword in the hope that we would be able to later devise and add a new declaration that would provide a more intuitive and direct means to specify simple records. That never happened–`record` remains reserved and unused to this day.

[36] It isn't available in any other significant language either so far as I know, so this is not so much a criticism of BLISS in particular as of programming language design in general.

The bind declaration, on the other hand, does get a lot of regular use. It is interesting to note the (very) approximate frequencies of the several ways in which the bind declaration can be used, which are shown here[37]:

Percentage of Occurrence	Purpose
60	Short name for component of aggregate
20	Name for generated static data
10	"Type" change (often after dynamically determining the kind of a polymorphic parameter)
10	Capture value (often for a macro argument to assure single evaluation semantics)

With the exception of declaring a name for initialized static storage, the other uses could be accomplished in other ways if bind were not available–but bind does provide a simple and intuitive mechanism that becomes used in most personal coding styles.

9.4 The Dot Operator

The dot operator usually throws new users for a loop for a while, but this gradually subsides. "Missing dot" bugs make good coffee stand conversation, but in fact don't happen all that often and when they do they tend to have dramatic easy to find effects rather than subtle ones (not always, but usually).

Overly enthusiastic new users sometimes try to code things with two or even three dots (fetches) in a row.[38] While not necessarily wrong, experience teaches that this is a definite stylistic error that should usually be avoided.

After these lessons are learned, using the dot becomes a routine part of one's coding practice–often like indentation–that one tends to do rather automatically while thinking about larger issues[39].

[37] This data is based on actual counts from a sampling of modules from GEM, but cannot be considered very "scientific"; nor does it suggest how often more than one purpose is involved.

[38] One colleague reports that there was a case in a certain piece of VMS software (prior to V5.5) where five consecutive dots were used; further, not only was it correct but it was the clearest way to document exactly what was being done!

[39] Some people liken this to learning to drive a shift transmission car when previously only used to an automatic transmission.

This surely sounds like a "damned by faint criticism" sort of defense–and in a way it is. On the other hand, the dot operator interacts with other aspects of BLISS, notably structures, typelessness, pervasive expressions, even map and bind declarations as well as parameter passing, in ways that would make it very hard to eliminate. Eliminating the dot operator would definitely require compensating changes in many of these other parts of the language.

9.5 No Run-Time Library

The goal of not requiring a run-time support library has always presented a naively attractive ideal but is overstated in practice. Even BLISS-10 had a helper module that needed to be linked with any module that used the BLISS co-routine construct. BLISS-11 had run-time routines needed for exception handling. Later BLISSes for RISC-oriented processors used run-time routines for fetch and assignment of dynamically sized fields, many of the character handling functions, and so on.

A more pragmatic and realistic goal statement for BLISS (or any other "low-level system language") might better be something like the following:

> No need for a run-time support library for the common and predominantly used language constructs. In addition, the implementation must provide simple and effective means to support a programmer intention to avoid constructs that do require separate run-time support.

9.6 No `goto`

This characteristic was a wonderful idea!

BLISS-10 eliminated the `goto` *and* labels, and compensated by including way too many forms of exit constructs. BLISS-11 corrected this mistake, and the resulting formulation of `leave` has been carried forward without any perceived need for reconsideration.

The experience of this author has been typical: whenever a piece of code gets to the point that it is tempting to use a `goto`, there is virtually guaranteed to be a reformulation without using `goto` that is cleaner, clearer and more elegant.

It is interesting to note that of the languages designed since BLISS, only Java [1] has followed suit. (It is about time!)

9.7 Expression Language

The expression language characteristic was often highly touted in the early years of BLISS. While there is a certain conceptual elegance that results, in practice this characteristic is not much exploited.

The most common applications use the `if-then-else` expression, for example, in something like the `and_then` and `or_else` macros described earlier; occasionally there is some analogous use of the `case` expression. Examples using loops (taking advantage of the value of `leave`), however, tend not to work well on human factors grounds: the value computed tends to be visually lost in the surrounding control constructs and too far removed from where it will be used; an explicit assignment to a temporary variable often seems to work better.

It is interesting to compare `leave` with `return` in this respect. Perhaps it is the prominence of the containing construct, so that it is quite clear where the value is used, that makes `return` so widely accepted.

On balance, the expression characteristic of BLISS was not terribly important.

9.8 Compiletime Language

The creation of a comprehensive compiletime facility was not, of course, one of the original BLISS goals; rather, that capability began to emerge in the form of more flexible macros in BLISS-11 and really came to fruition in the second generation of BLISSes created at DEC.

It is not possible to stress enough just how highly the compiletime facility is regarded here at DEC. It has been used perhaps most importantly in any number of schemes for table creation, dumpers, and other tools that, because they are an integrated part of the language and the code being created, contribute to uniformity, consistency and ultimately, reliability. (The exploitation of this facility on the VAX Ada project as described in Section 4.2 is but one example–there are many others.)

No other commercial quality language comes close to matching this capability.

9.9 Relationship to Optimization Technology

In his Ph.D. thesis, Geschke [23] showed how to exploit two consequences of the lack of `goto` in BLISS:

- The abstract syntax representation of a program that results from parsing can also serve directly as a flow representation for the purposes of global flow analysis
- Flow analysis algorithms can be simplified because the program flow graph is inherently always reducible.

These algorithms were first embodied in the BLISS-11 compiler and then propagated into all of DEC's second generation compilers. As a result, the early BLISS compilers benefited from very high quality global optimization at rather modest costs in terms of execution time and memory size. This did much to make BLISS code quality good enough to ease the greatest part of the code quality based concerns about using BLISS at a time when even "large" computers were puny by today's standards.

Since GEM is intended to serve many languages and programming styles, including the worst classic "dusty deck" FORTRAN applications around, it uses more general and more resource intensive techniques. More frugal approaches are no longer needed.

9.10 BLISS and Portability

It is appropriate to reiterate an assessment of BLISS and portability offered in 1980 [6]:

> *There is no doubt that the low-level data semantics of BLISS results in an initially higher development cost than in more inherently transportable languages. However, as we [DEC] evolved the techniques and conventions described here it became clear that many of the anticipated difficulties were more apparent than real. Indeed, the first problem is a psychological one: it is easy to dwell on the problems and never seriously explore the possible solutions.*
>
> *The essential point is this: in BLISS, the degree of storage [or other] inefficiency due to transportability goals is constrained only by the intrinsic characteristics of the problem*

> *to be solved and the underlying machine and operating system architectures to be used, and not by limitations of the programming language. In this sense, BLISS is a better language for transportable programming than more traditional high-level languages precisely because of its machine oriented character. This advantage does not come without a cost, to be sure. But the opportunity for simultaneous transportability and optimality is unusual.*

Twenty years later that assessment remains equally valid. However, if you don't have a BLISS implementation for one of your target environments, the question is moot.

9.11 Language Evolution

There is no doubt that the amount of language evolution that took place during the creation of DEC's second generation of BLISSes was far greater than anyone (including those of us doing it) had anticipated.

About the only way to assess whether we "got it right" is to look at the degree of evolution that took place as BLISS was extended to later machine architectures that had not been considered as part of the original design. Since that later evolution was both small and almost entirely within the framework established to allow extension, it appears fair to conclude that by and large we did get it right–at least within the constraint of being typeless.

9.12 C

It is inevitable that some kind of "BLISS vs C" comparison or evaluation should be attempted[40]. They both emerged about the same time from the same historical context and they shared many points of view.

When all is said and done, perhaps the most important difference is that the designers of C chose to keep a mostly traditional data type model that was both simple and "good enough". In particular, scalars that happen to be larger that the machine word size (not the least unusual for floating point some 30 years ago) and records (`structs`) could be declared simply and used in straightforward ways. The BLISS typeless data model, based on all data being of the underlying machine word

[40] However, quite honestly, I am probably one of the worst candidates to offer an objective perspective. After all, C "won", right?

Language Design is a Little Bit Like Sex

- It is universally appealing–everyone wants to get into the act
- It is easy to get started–you need no special equipment and only a little training
- It is easy to do badly, hard to do well, and hardest of all to know the difference
- Language design between consenting adults in the privacy of their own work is one thing, but be careful what you turn loose on the world
- You won't understand what you have (really) created until long afterward

–R. F. Brender, "What Did DEC Do to BLISS and Why?"
CMU Computer Science Department Lecture, 1979

---

size, definitely made some of these simple things not so simple. As recorded here, BLISS gradually evolved a whole lot of machinery to help mitigate those problems; the result is effective but far from simple.

One other consideration is probably crucial as well: C originally came complete with an operating system whose vitality (and economics) strongly aided its spread (propelled by what might be considered the original "open source" movement). BLISS had no such vehicle.

I will share one blatant opinion: C sure could use some decent compiletime capabilties.

10 Summary

From a highly machine specific implementation language on the PDP-10, BLISS evolved into a family of highly compatible and portable languages and compilers for numerous environments:

- PDP-10: TOPS-10 (DECsystem-10) and TOPS-20 (DECSYSTEM-20)
- PDP-11: RSX-11M and RSX-11D
- VAX: VMS
- MIPS: Ultrix
- Alpha: VMS, Tru64 UNIX, Windows NT and Linux
- Intel IA-32: Windows 95/98 and Windows NT

Along the way there have been other at least short term off-shoots for the IBM 360/370 [2] and Honeywell 6000 [24][41]. Each BLISS remains true to an initial goal to allow and support low level machine specific and idiosyncratic access and control *when wanted and needed*, while providing mechanisms for portability as well.

During the Golden Years, BLISS enjoyed widespread support and use within DEC. And while there were a modest number of enthusiastic users and supporters outside of DEC, DEC never promoted BLISS beyond making it available as a product and never sought to port it beyond its own product lines. As a result, BLISS never became very well known in the general computing community.

Ironically, as DEC itself faltered in the 1990s, the divestiture of software products written in BLISS did more to spread knowledge and use of BLISS outside DEC (now Compaq) than ever before. EDS, Oracle, Computer Associates and others now have ongoing products that depend on or use BLISS to a significant degree. No significant new project has chosen BLISS for its implementation language for at least five years.

While the language is clearly near the end of its life, it is continuing to expand to new architectures and organizations, though only as a captive tool.

[41] And even unsubstantiated rumors of BLISSes for one or more Russian machines [24].

Acknowledgements

It is truly impossible to credit everyone who has made a significant contribution to the development and evolution of BLISS over the last 30 years. However, some contributions really stand out and deserve special mention.

At CMU, Bill Wulf was the creator of the language, whose inspiration, coding and leadership brought the ideas to reality.

Chuck Geschke, Steve Hobbs, Rich Johnsson, Bruce Leverett, and Dave Wile contributed code, generated language ideas and infused new technology into BLISS while earning their Ph.D.s during the CMU years.

At DEC, Gordon Bell had the courage to point DEC, and invest heavily, toward the use of a high level systems implementation language at a time when many thought it risky or foolish or both.

Marty Jack was a member of the team that created the second generation of BLISSes, whose superb coding skills and outstanding productivity got the new implementations solidly underway quickly. Topher Cooper proposed and initially led the preprocessor strategy that solved our language bootstrapping problem. Ike Nassi provided lots of language ideas for making BLISS as portable as possible.

Rich Grove was the first major BLISS-11 user at DEC when he led the compiler part of the PDP-11 FORTRAN IV-PLUS product; he continued to use and promote BLISS-32 on VAX FORTRAN and Ada product developments, and continues to use BLISS-32 and -64 as Technical Director of the GEM Retargetable Backend Code Generator project which is now used in all of Compaq's compilers for Alpha and other architectures.

Addendum – What's In a Name

Choosing the right names to use in this document sometimes has been difficult because different names have been used for the same things over time. Here is a key that may be of use.

BLISS was the original name of the language invented specifically for the PDP-10. That name changed to BLISS-10 to differentiate it from BLISS-11 for the PDP-11. BLISS-VAX was the original name used for the version of BLISS for the VAX. That name changed to BLISS-32 to go along with BLISS-16 and BLISS-36 which referred to the second generation successors to BLISS-11 and BLISS-10, respectively. BLISS-64 came into use later for the Alpha version of BLISS. Where there are several variants of BLISS for the same word size (for example, BLISS-32 for VAX, MIPS, Alpha and IA-32), no naming convention has been formulated or emerged so you may need to infer from context if it matters. BLISS now refers collectively to all BLISS languages, although it may or may not be intended to include the original BLISS-10 and BLISS-11 and/or other non-DEC variants.

DEC is the name by which Digital Equipment Corporation was generally known, until someone decided that DIGITAL was better. Digital Equipment Corporation no longer exists even as a legal entity; it was purchased by Compaq Computer Corporation in June 1998 and is now completely merged into Compaq.

PDP is a trademark used by DEC for many years, and was often included in combination with a number as (part of) the name of a computer product. The PDP-1 was DEC's first, while the PDP-2 and PDP-3 never came to market. The PDP-4, PDP-7, PDP-9, and PDP-15 were very similar to the PDP-1 in many ways, and were strongly upward compatible among themselves–all had 18-bit processor architectures. The PDP-5, PDP-8 and PDP-12 made a 12-bit processor family. The PDP-6 was the first 36-bit processor and was succeeded by the PDP-10. There was never really a PDP-20 but the name was sometimes used to refer to the variant of the PDP-10 used for the TOPS-20 operating system. The PDP-11 referred to the 16-bit processor family, of which the PDP-11/20 was the first; it was succeeded by the PDP-11/40, PDP-11/45 and PDP-11/70 (among others).

TOPS-10 referred to the operating system for the PDP-10 and at some point the combination became called the DECsystem-10. TOPS-20 was a later alternative operating system for a variant of the PDP-10 and that combination was called the DECSYSTEM-20[42].

DOS-11, RT-11, RSX-11D, RSX-11M, and RSTS-11 (among others) were operating systems for the PDP-11.

[42] Yes, the capitalization was different!

VAX, which stood for Virtual Address eXtension, was the internal code name for the 32-bit architecture developed to succeed the PDP-11 architecture. It became VAX-11 for the first few members of the product, to emphasize "cultural compatibility" with the PDP-11 in the form of common data types, microcode emulation of usermode PDP-11 instructions and software emulation of the RSX-11M operating system within VMS. MicroVAX was used for some low-end but completely compatible variants. VAX superseded VAX-11 once all hardware support for the PDP-11 was dropped from the line.

VMS, Virtual Memory System, refers to the operating system for the VAX-11. OpenVMS is a newer name for exactly the same software.

Alpha is the name of the 64-bit architecture that followed the VAX. AXP was used for a brief time, either alone or following Alpha, but is no longer used.

DEC OSF/1, DIGITAL UNIX and now Compaq Tru64 UNIX are successive names for the same operating system on Alpha systems.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, 1996. Lisa Friendly, Series Editor.

[2] A BLISS[-3] LANGUAGE PRIMER [for the ibm 360/370]. Technical report, Advance Program Technology Department, Bell Laboratories, December 1973. Preliminary Version.

[3] David S. Blickstein, Peter W. Craig, Caroline S. Davidson, R. Neil Faiman, Jr., Kent D. Glossop, Richard B. Grove, Steven O. Hobbs, and William B. Noyce. The GEM optimizing compiler system. *Digital Technical Journal*, 4(4), Special Issue 1992.

[4] R. F. Brender. BLISS-11 and FORTRAN IV-PLUS: A case study in the application of a high level cross-compiler to product development. In J. R. Bell and C. G. Bell, editors, *Minicomputer Software*, pages 53–69. North Holland, 1976.

[5] R. F. Brender. A survey of BLISS-16, BLISS-32 and BLISS-36. In *Proceedings of the Digital Equipment Corporation Users Society*, volume 4, pages 1113–1127. DECUS, April 1978.

[6] Ronald F. Brender. Generation of BLISSes. *IEEE Transactions on Software Engineering*, pages 553–562, November 1980. Also as Technical Report CMU-CS-79-125, Carnegie-Mellon University, May 1979.

[7] Ronald F. Brender, Bevin R. Brett, and Charles Z. Mitchell. Pragmatics in the development of VAX Ada. *Digital Technical Journal*, (6):91–100, February 1988.

[8] Burroughs Corporation, Detroit, MI. *Burroughs B5500 Extended Algol.*

[9] Danny Cohen. On holy wars and a plea for peace. *IEEE Computer Magazine*, 14:48–54, October 1981. Also at `http://www.op.net/docs/RFCs/ien-137`.

[10] Digital Equipment Corporation. 1976 Annual Report.

[11] Digital Equipment Corporation. BLISS-11 Reference Manual. Technical report, DECUS Program Library, 1974.

[12] Digital Equipment Corporation. *BLISS-10 Programmer's Reference Manual*, 1974. Order Number DEC-10-LBRMA-A-D.

[13] Digital Equipment Corporation. *VAX-11 Software Engineering Manual*, 1977. Later superceded by [20].

[14] Digital Equipment Corporation. *VAX-11/780 Architecture Handbook, Vol. I*, 1977.

[15] Digital Equipment Corporation. *BLISS-32 Language Guide*, 1978. Order Number AA-H323A-TE.

[16] Digital Equipment Corporation. *BLISS Language Guide*, 1978. (Restricted to internal distribution).

[17] Digital Equipment Corporation. *BLISS Primer*, 1979. Bound as 3 volumes: Introductory [Modules], Intermediate [Modules], and Readings.

[18] Digital Equipment Corporation. *BLISS Language Guide*, 2nd edition, January 1980. Order Number AA-H275B-TK.

[19] Digital Equipment Corporation. *Micro-BLISS USER'S GUIDE, Preliminary Version*, January 1981.

[20] Digital Equipment Corporation. *Software Engineering Manual*, 1988. Document Identifier A-DG-ELEN571-00-0 Rev. A, Internal corporate standard.

[21] Digital Equipment Corporation. *PDP-10 Architecture Reference Manual*, circa 1965.

[22] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968. Letter to the Editor.

[23] Charles M. Geschke. *Global Program Optimizations*. PhD thesis, Carnegie-Mellon University, 1972.

[24] Steve Hobbs. Personal communication.

[25] Richard K. Johnsson. *An Approach to Global Register Allocation*. PhD thesis, Carnegie-Mellon University, 1976.

[26] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Inc, New York, 1992.

[27] B. W. Leverett, P. J. Knueven, and P. G. Hibbard. Portability Techniques for BLISS Programs. Technical report, Carnegie-Mellon University, Computer Science Department, November 1977.

[28] M. P. Marks. Transportable programming with BLISS. In *Proceedings of the Digital Equipment Corporation Users Society*, pages 1109–1112. Digital Equipment Corporation Users Society, April 1978.

[29] MIT Project MAC. *EPL Reference Manual*, April 1965.

[30] Bill Parke, 2000. Personal communication.

[31] Martin Richards. *BCPL Reference Manual*. MIT Project MAC, 1967.

[32] Tim Robinson. 99 bottles of beer on the wall, December 1996. `http://www.ionet.net/~timtroyr/funhouse/beer.html`.

[33] Bill Segal. Personal communication.

[34] Bill Segal. BLISS PROGRAM PLAN, April 1976. Development Methods Department, Digital Equipment Corporation, internal document.

[35] Spring Joint Computer Conference. *BCPL: A tool for compiler writing and systems programming*, 1969.

[36] Charles B. Weinstock. *Dynamic storage allocation techniques.* PhD thesis, Carnegie-Mellon University, 1976.

[37] D. S. Wile and C. M. Geschke. Efficient data accessing in the programming language BLISS. In *SIGPLAN Symposium on Data Structures Proceedings.* Association for Computing Machinery, February 1971.

[38] David S. Wile. *A generative, nested-sequential basis for general purpose programming languages.* PhD thesis, Carnegie-Mellon University, 1973.

[39] N. Wirth. PL/360, a programming language for the 360 computers. *Journal of the ACM*, 15(1), January 1968.

[40] W. Wulf. Programming without the goto. In *Proceedings of the IFIP Congress*, pages 408–413. North Holland, October 1971. Also included in [17], Vol III: Readings.

[41] W. Wulf. Bliss/11 dec mini course [sic]. Unpublished lecture notes, 1972.

[42] W. Wulf. Systems for systems implementation: Some experiences from BLISS. In *AFIPS Fall Joint Computer Conference*, 1972.

[43] W. Wulf, March 2000. Personal communication.

[44] W. Wulf, C. Geschke, D. Wile, and J. Apperson. Reflections on a systems programming language. In *Proceedings of a SIGPLAN Symposium on Languages for Systems Implementation*, pages 42–49, October 1971.

[45] W. A. Wulf. Letter to Steve Gutz, Development Methods Manager, September 16, 1976.

[46] W. A. Wulf, R. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler.* Number 2 in Programming Languages Series. American Elsevier, New York, 1975. Thomas E. Cheatham, Ed.

[47] W. A. Wulf, D. B. Russell, and A. N. Habermann. BLISS: A language for systems programming. *Communications of the ACM*, 14(12):780–790, December 1971.